Zebra - User's Guide and Reference

Copyright © 1994-2009 Index Data

COLLABORATORS

	TITLE : Zebra - User's Guide and Reference	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Adam Dickmeiss, Heikki Levanto, Marc Cromme, Mike Taylor, and Sebastian Hammer	November 5, 2009	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**

1.1 Overview . . . 1

1.2 Zebra Features Overview . . . 2

1.2.1 Zebra Document Model . . . 2

1.2.2 Zebra Search Features . . . 2

1.2.3 Zebra Index Scanning . . . 2

1.2.4 Zebra Document Presentation . . . 2

1.2.5 Zebra Sorting and Ranking . . . 2

1.2.6 Zebra Live Updates . . . 2

1.2.7 Zebra Networked Protocols . . . 2

1.2.8 Zebra Data Size and Scalability . . . 2

1.2.9 Zebra Supported Platforms . . . 2

1.3 References and Zebra based Applications . . . 2

1.3.1 Koha free open-source ILS . . . 4

1.3.2 Kete Open Source Digital Library and Archiving software . . . 9

1.3.3 Emilda open source ILS . . . 9

1.3.4 ReIndex.Net web based ILS . . . 9

1.3.5 DADS - the DTV Article Database Service . . . 9

1.3.6 Infonet Eprints . . . 10

1.3.7 Alvis . . . 10

1.3.8 ULS (Union List of Serials) . . . 10

1.3.9 NLI-Z39.50 - a Natural Language Interface for Libraries . . . 10

1.3.10 Various web indexes . . . 11

1.4 Support . . . 11

2 Installation **13**

2.1 UNIX . . . 13

2.2 GNU/Debian . . . 14

2.2.1 GNU/Debian Linux on i686 Platform . . . 14

2.2.2 Ubuntu/Debian and GNU/Debian on other platforms . . . 15

2.3 WIN32 . . . 15

2.4 Upgrading from Zebra version 1.3.x . . . 17

3 Tutorial **19**

3.1 A first OAI indexing example . . . 19

3.2 Searching the OAI database by web service . . . 20

3.3 Presenting search results in different formats . . . 20

3.4 More interesting searches . . . 21

3.5 Investigating the content of the indexes . . . 22

3.6 Setting up a correct SRU web service . . . 22

3.7 Searching the OAI database by Z39.50 protocol . . . 23

4 Overview of Zebra Architecture **27**

4.1 Local Representation . . . 27

4.2 Main Components . . . 27

4.2.1 Core Zebra Libraries Containing Common Functionality . . . 27

4.2.2 Zebra Indexer . . . 28

4.2.3 Zebra Searcher/Retriever . . . 28

4.2.4 YAZ Server Frontend . . . 28

4.2.5 Record Models and Filter Modules . . . 29

4.2.5.1 DOM XML Record Model and Filter Module . . . 29

4.2.5.2 ALVIS XML Record Model and Filter Module . . . 29

4.2.5.3 GRS-1 Record Model and Filter Modules . . . 30

4.2.5.4 TEXT Record Model and Filter Module . . . 30

4.3 Indexing and Retrieval Workflow . . . 30

4.4 Retrieval of Zebra internal record data . . . 31

5 Query Model **35**
5.1 Query Model Overview . . . 35
5.1.1 Query Languages . . . 35
5.1.1.1 Prefix Query Format (PQF) . . . 35
5.1.1.2 Common Query Language (CQL) . . . 35
5.1.2 Operation types . . . 35
5.1.2.1 Explain Operation . . . 36
5.1.2.2 Search Operation . . . 36
5.1.2.3 Scan Operation . . . 36
5.2 RPN queries and semantics . . . 36
5.2.1 RPN tree structure . . . 36
5.2.1.1 Attribute sets . . . 37
5.2.1.2 Boolean operators . . . 37
5.2.1.3 Atomic queries (APT) . . . 38
5.2.1.4 Named Result Sets . . . 39
5.2.1.5 Zebra's special access point of type 'string' . . . 40
5.2.1.6 Zebra's special access point of type 'XPath' for GRS-1 filters . . . 40
5.2.2 Explain Attribute Set . . . 42
5.2.2.1 Use Attributes (type = 1) . . . 42
5.2.2.2 Explain searches with yaz-client . . . 42
5.2.3 BIB-1 Attribute Set . . . 43
5.2.3.1 Use Attributes (type 1) . . . 43
5.2.4 Zebra general Bib1 Non-Use Attributes (type 2-6) . . . 44
5.2.4.1 Relation Attributes (type 2) . . . 44
5.2.4.2 Position Attributes (type 3) . . . 45
5.2.4.3 Structure Attributes (type 4) . . . 46
5.2.4.4 Truncation Attributes (type = 5) . . . 47
5.2.4.5 Completeness Attributes (type = 6) . . . 48
5.3 Extended Zebra RPN Features . . . 49
5.3.1 Zebra specific retrieval of all records . . . 49
5.3.2 Zebra specific Search Extensions to all Attribute Sets . . . 50
5.3.2.1 Zebra Extension Embedded Sort Attribute (type 7) . . . 50
5.3.2.2 Zebra Extension Rank Weight Attribute (type 9) . . . 51
5.3.2.3 Zebra Extension Term Reference Attribute (type 10) . . . 51

5.3.2.4 Local Approximative Limit Attribute (type 11) . . . 51
5.3.2.5 Global Approximative Limit Attribute (type 12) . . . 52
5.3.3 Zebra specific Scan Extensions to all Attribute Sets . . . 52
5.3.3.1 Zebra Extension Result Set Narrow (type 8) . . . 52
5.3.3.2 Zebra Extension Approximative Limit (type 12) . . . 53
5.3.4 Zebra special IDXPATH Attribute Set for GRS-1 indexing . . . 53
5.3.4.1 IDXPATH Use Attributes (type = 1) . . . 53
5.3.5 Mapping from PQF atomic APT queries to Zebra internal register indexes . . . 54
5.3.5.1 Mapping of PQF APT access points . . . 55
5.3.5.2 Mapping of PQF APT structure and completeness to register type . . . 56
5.3.6 Zebra Regular Expressions in Truncation Attribute (type = 5) . . . 58
5.4 Server Side CQL to PQF Query Translation . . . 59

6 Administrating Zebra **61**
6.1 Record Types . . . 61
6.2 The Zebra Configuration File . . . 62
6.3 Locating Records . . . 64
6.4 Indexing with no Record IDs (Simple Indexing) . . . 64
6.5 Indexing with File Record IDs . . . 65
6.6 Indexing with General Record IDs . . . 65
6.7 Register Location . . . 66
6.8 Safe Updating - Using Shadow Registers . . . 67
6.8.1 Description . . . 67
6.8.2 How to Use Shadow Register Files . . . 68
6.9 Relevance Ranking and Sorting of Result Sets . . . 69
6.9.1 Overview . . . 69
6.9.2 Static Ranking . . . 69
6.9.3 Dynamic Ranking . . . 70
6.9.3.1 Dynamically ranking using PQF queries with the 'rank-1' algorithm . . . 70
6.9.3.2 Dynamically ranking CQL queries . . . 71
6.9.4 Sorting . . . 72
6.10 Extended Services: Remote Insert, Update and Delete . . . 73
6.10.1 Extended services in the Z39.50 protocol . . . 74
6.10.2 Extended services from yaz-client . . . 75
6.10.3 Extended services from yaz-php . . . 76
6.10.4 Extended services debugging guide . . . 77

7 DOM XML Record Model and Filter Module **79**
7.1 DOM Record Filter Architecture . . . 79
7.2 DOM XML filter pipeline configuration . . . 80
7.2.1 Input pipeline . . . 82
7.2.2 Extract pipeline . . . 82
7.2.3 Store pipeline . . . 82
7.2.4 Retrieve pipeline . . . 82
7.2.5 Canonical Indexing Format . . . 83
7.2.5.1 Processing-instruction governed indexing format . . . 83
7.2.5.2 Magic element governed indexing format . . . 83
7.2.5.3 Semantics of the indexing formats . . . 84
7.3 DOM Record Model Configuration . . . 86
7.3.1 DOM Indexing Configuration . . . 86
7.3.2 DOM Indexing MARCXML . . . 87
7.3.3 DOM Indexing Wizardry . . . 89
7.3.4 Debuggig DOM Filter Configurations . . . 90

8 ALVIS XML Record Model and Filter Module **91**
8.1 ALVIS Record Filter . . . 91
8.1.1 ALVIS Internal Record Representation . . . 92
8.1.2 ALVIS Canonical Indexing Format . . . 92
8.2 ALVIS Record Model Configuration . . . 94
8.2.1 ALVIS Indexing Configuration . . . 94
8.2.2 ALVIS Exchange Formats . . . 96
8.2.3 ALVIS Filter OAI Indexing Example . . . 97

9 GRS-1 Record Model and Filter Modules **99**
9.1 GRS-1 Record Filters . . . 99
9.1.1 GRS-1 Canonical Input Format . . . 100
9.1.1.1 Record Root . . . 101
9.1.1.2 Variants . . . 101
9.1.2 GRS-1 REGX And TCL Input Filters . . . 102
9.2 GRS-1 Internal Record Representation . . . 103
9.2.1 Tagged Elements . . . 104
9.2.2 Variants . . . 104

9.2.3 Data Elements . . . 105
9.3 GRS-1 Record Model Configuration . . . 105
9.3.1 The Abstract Syntax . . . 105
9.3.2 The Configuration Files . . . 106
9.3.3 The Abstract Syntax (.abs) Files . . . 106
9.3.4 The Attribute Set (.att) Files . . . 108
9.3.5 The Tag Set (.tag) Files . . . 109
9.3.6 The Variant Set (.var) Files . . . 110
9.3.7 The Element Set (.est) Files . . . 111
9.3.8 The Schema Mapping (.map) Files . . . 112
9.3.9 The MARC (ISO2709) Representation (.mar) Files . . . 113
9.4 GRS-1 Exchange Formats . . . 113
9.5 Extended indexing of MARC records . . . 113
9.5.1 The index-formula . . . 114
9.5.2 Notation of *index-formula* for Zebra . . . 115
9.5.2.1 Examples . . . 116

10 Field Structure and Character Sets **117**
10.1 The default.idx file . . . 117
10.2 Charmap Files . . . 119
10.3 ICU Chain Files . . . 122

11 Reference **125**
11.1 zebraidx . . . 125
11.2 zebrasrv . . . 127
11.3 idzebra-config . . . 136

A License **137**

B GNU General Public License **139**
B.1 Preamble . . . 139
B.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 140
B.2.1 Section 0 . . . 140
B.2.2 Section 1 . . . 140
B.2.3 Section 2 . . . 140

B.2.4 Section 3 . . . 141
B.2.5 Section 4 . . . 142
B.2.6 Section 5 . . . 142
B.2.7 Section 6 . . . 142
B.2.8 Section 7 . . . 142
B.2.9 Section 8 . . . 143
B.2.10 Section 9 . . . 143
B.2.11 Section 10 . . . 143
B.2.12 NO WARRANTY Section 11 . . . 143
B.2.13 Section 12 . . . 143
B.3 How to Apply These Terms to Your New Programs . . . 144

C About Index Data and the Zebra Server **145**

List of Figures

7.1 DOM XML filter architecture 80

List of Tables

1.1 Zebra document model . . . 2
1.2 Zebra search functionality . . . 3
1.3 Zebra index scanning . . . 4
1.4 Zebra document presentation . . . 5
1.5 Zebra sorting and ranking . . . 6
1.6 Zebra live updates . . . 6
1.7 Zebra networked protocols . . . 7
1.8 Zebra data size and scalability . . . 8
1.9 Zebra supported platforms . . . 8

4.1 Special Retrieval Elements . . . 31

5.1 Attribute sets predefined in Zebra . . . 37
5.2 Boolean operators . . . 38
5.3 Atomic queries (APT) . . . 39
5.4 Relation Attributes (type 2) . . . 44
5.5 Position Attributes (type 3) . . . 45
5.6 Structure Attributes (type 4) . . . 46
5.7 Truncation Attributes (type 5) . . . 47
5.8 Completeness Attributes (type = 6) . . . 49
5.9 Zebra Search Attribute Extensions . . . 50
5.10 Zebra Scan Attribute Extensions . . . 52
5.11 Zebra specific IDXPATH Use Attributes (type 1) . . . 54
5.12 Access point name mapping . . . 55
5.13 Structure and completeness mapping to register types . . . 57
5.14 Regular Expression Operands . . . 58
5.15 Regular Expression Operators . . . 58

6.1 Extended services Z39.50 Package Fields 74

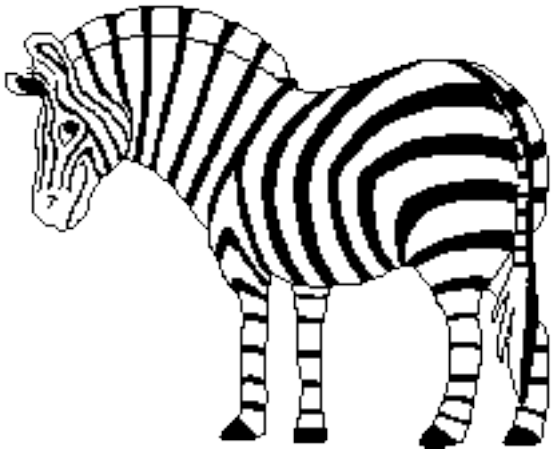
7.1 DOM XML filter pipelines overview 81

10.1 Character maps predefined in Zebra 120

Abstract

Zebra is a free, fast, friendly information management system. It can index records in XML, SGML, MARC, e-mail archives and many other formats, and quickly find them using a combination of boolean searching and relevance ranking. Search-and-retrieve applications can be written using APIs in a wide variety of languages, communicating with the Zebra server using industry-standard information-retrieval protocols or web services.

This manual explains how to build and install Zebra, configure it appropriately for your application, add data and set up a running information service. It describes version 2.0.43 of Zebra.



Chapter 1

Introduction

1.1 Overview

Zebra is a free, fast, friendly information management system. It can index records in XML/SGML, MARC, e-mail archives and many other formats, and quickly find them using a combination of boolean searching and relevance ranking. Search-and-retrieve applications can be written using APIs in a wide variety of languages, communicating with the Zebra server using industry-standard information-retrieval protocols or web services.

Zebra is licensed Open Source, and can be deployed by anyone for any purpose without license fees. The C source code is open to anybody to read and change under the GPL license.

Zebra is a networked component which acts as a reliable Z39.50 server for both record/document search, presentation, insert, update and delete operations. In addition, it understands the SRU family of webservices, which exist in REST GET/POST and truly SOAP flavors.

Zebra is available as MS Windows 2003 Server (32 bit) self-extracting package as well as GNU/Debian Linux (32 bit and 64 bit) precompiled packages. It has been deployed successfully on other Unix systems, including Sun Sparc, HP Unix, and many variants of Linux and BSD based systems.

http://www.indexdata.com/zebra/ http://ftp.indexdata.dk/pub/zebra/win32/ http://ftp.indexdata.dk/pub/zebra/-debian/

Zebra is a high-performance, general-purpose structured text indexing and retrieval engine. It reads records in a variety of input formats (e.g. email, XML, MARC) and provides access to them through a powerful combination of boolean search expressions and relevance-ranked free-text queries.

Zebra supports large databases (tens of millions of records, tens of gigabytes of data). It allows safe, incremental database updates on live systems. Because Zebra supports the industry-standard information retrieval protocol, Z39.50, you can search Zebra databases using an enormous variety of programs and toolkits, both commercial and free, which understand this protocol. Application libraries are available to allow bespoke clients to be written in Perl, C, C++, Java, Tcl, Visual Basic, Python, PHP and more - see the ZOOM web site for more information on some of these client toolkits.

This document is an introduction to the Zebra system. It explains how to compile the software, how to prepare your first database, and how to configure the server to give you the functionality that you need.

1.2 Zebra Features Overview

1.2.1 Zebra Document Model

Feature	Availability	Notes	Reference
Complex semi-structured Documents	XML and GRS-1 Documents	Both XML and GRS-1 documents exhibit a DOM like internal representation allowing for complex indexing and display rules	Chapter 8 and Chapter 9
Input document formats	XML, SGML, Text, ISO2709 (MARC)	A system of input filters driven by regular expressions allows most ASCII-based data formats to be easily processed. SGML, XML, ISO2709 (MARC), and raw text are also supported.	Section 4.2.5
Document storage	Index-only, Key storage, Document storage	Data can be, and usually is, imported into Zebra's own storage, but Zebra can also refer to external files, building and maintaining indexes of "live" collections.	

Table 1.1: Zebra document model

1.2.2 Zebra Search Features

1.2.3 Zebra Index Scanning

1.2.4 Zebra Document Presentation

1.2.5 Zebra Sorting and Ranking

1.2.6 Zebra Live Updates

1.2.7 Zebra Networked Protocols

1.2.8 Zebra Data Size and Scalability

1.2.9 Zebra Supported Platforms

1.3 References and Zebra based Applications

Zebra has been deployed in numerous applications, in both the academic and commercial worlds, in application domains as diverse as bibliographic catalogues, Geo-spatial information, structured vocabulary

Feature	Availability	Notes	Reference
Query languages	CQL and RPN/PQF	The type-1 Reverse Polish Notation (RPN) and its textual representation Prefix Query Format (PQF) are supported. The Common Query Language (CQL) can be configured as a mapping from CQL to RPN/PQF	Section 5.1.1.1 and Section 5.4
Complex boolean query tree	CQL and RPN/PQF	Both CQL and RPN/PQF allow atomic query parts (APT) to be combined into complex boolean query trees	Section 5.2.1
Field search	user defined	Atomic query parts (APT) are either general, or directed at user-specified document fields	Section 5.2.1.3, Section 5.2.1.5, Section 5.2.3.1, and Section 5.3.4.1
Data normalization	user defined	Data normalization, text tokenization and character mappings can be applied during indexing and searching	Chapter 10
Predefined field types	user defined	Data fields can be indexed as phrase, as into word tokenized text, as numeric values, URLs, dates, and raw binary data.	Section 10.2 and Section 5.3.5.2
Regular expression matching	available	Full regular expression matching and "approximate matching" (e.g. spelling mistake corrections) are handled.	Section 5.3.6
Term truncation	left, right, left-and-right	The truncation attribute specifies whether variations of one or more characters are allowed between search term and hit terms, or not. Using non-default truncation attributes will broaden the document hit set of a search query.	Section 5.2.4.4
Fuzzy searches	Spelling correction	In addition, fuzzy searches are implemented, where one spelling mistake in search terms is matched	Section 5.2.4.4

Table 1.2: Zebra search functionality

Feature	Availability	Notes	Reference
Scan	term suggestions	`Scan` on a given named index returns all the indexed terms in lexicographical order near the given start term. This can be used to create drop-down menus and search suggestions.	Section 5.1.2.3 and Section 5.2.1.3
Facetted browsing	available	Zebra 2.1 and allows retrieval of facets for a result set.	Section 5.3.3
Drill-down or refine-search	partially	scanning in result sets can be used to implement drill-down in search clients	Section 5.3.3

Table 1.3: Zebra index scanning

browsing, government information locators, civic information systems, environmental observations, museum information and web indexes.

Notable applications include the following:

1.3.1 Koha free open-source ILS

Koha is a full-featured open-source ILS, initially developed in New Zealand by Katipo Communications Ltd, and first deployed in January of 2000 for Horowhenua Library Trust. It is currently maintained by a team of software providers and library technology staff from around the globe.

LibLime, a company that is marketing and supporting Koha, adds in the new release of Koha 3.0 the Zebra database server to drive its bibliographic database.

In early 2005, the Koha project development team began looking at ways to improve MARC support and overcome scalability limitations in the Koha 2.x series. After extensive evaluations of the best of the Open Source textual database engines - including MySQL full-text searching, PostgreSQL, Lucene and Plucene - the team selected Zebra.

"Zebra completely eliminates scalability limitations, because it can support tens of millions of records." explained Joshua Ferraro, LibLime's Technology President and Koha's Project Release Manager. "Our performance tests showed search results in under a second for databases with over 5 million records on a modest i386 900Mhz test server."

"Zebra also includes support for true boolean search expressions and relevance-ranked free-text queries, both of which the Koha 2.x series lack. Zebra also supports incremental and safe database updates, which allow on-the-fly record management. Finally, since Zebra has at its heart the Z39.50 protocol, it greatly improves Koha's support for that critical library standard."

Although the bibliographic database will be moved to Zebra, Koha 3.0 will continue to use a relational SQL-based database design for the 'factual' database. "Relational database managers have their strengths, in spite of their inability to handle large numbers of bibliographic records efficiently," summed up Ferraro, "We're taking the best from both worlds in our redesigned Koha 3.0.

See also LibLime's newsletter article Koha Earns its Stripes.

Feature	Availability	Notes	Reference
Hit count	yes	Search results include at any time the total hit count of a given query, either exact computed, or approximative, in case that the hit count exceeds a possible pre-defined hit set truncation level.	Section 5.3.2.4 and Section 6.2
Paged result sets	yes	Paging of search requests and present/display request can return any successive number of records from any start position in the hit set, i.e. it is trivial to provide search results in successive pages of any size.	
XML document transforma-tions	XSLT based	Record presentation can be performed in many pre-defined XML data formats, where the original XML records are on-the-fly transformed through any preconfigured XSLT transformation. It is therefore trivial to present records in short/full XML views, transforming to RSS, Dublin Core, or other XML based data formats, or transform records to XHTML snippets ready for inserting in XHTML pages.	Section 8.2.2
Binary record trans-formations	MARC, USMARC, MARC21 and MARCXML	post-filter record transformations	
Record Syntaxes		Multiple record syntaxes for data retrieval: GRS-1, SUTRS, XML, ISO2709 (MARC), etc. Records can be mapped between record syntaxes and schemas on the fly.	
Zebra internal metadata	yes	Zebra internal document metadata can be fetched in SUTRS and XML record syntaxes. Those are useful in client applications.	Section 4.4
Zebra internal raw record data	yes	Zebra internal raw, binary record data can be fetched in SUTRS and XML record syntaxes, leveraging %zebra; to a binary storage system	Section 4.4
Zebra internal record field data	yes	Zebra internal record field data can be fetched in SUTRS and XML record syntaxes. This makes very fast minimal record data displays possible.	Section 4.4

Table 1.4: Zebra document presentation

Feature	Availability	Notes	Reference
Sort	numeric, lexicographic	Sorting on the basis of alpha-numeric and numeric data is supported. Alphanumeric sorts can be configured for different data encodings and locales for European languages.	Section 6.9.4 and Section 5.3.2.1
Combined sorting	yes	Sorting on the basis of combined sorts e.g. combinations of ascending/descending sorts of lexicographical/numeric/date field data is supported	Section 6.9.4
Relevance ranking	TF-IDF like	Relevance-ranking of free-text queries is supported using a TF-IDF like algorithm.	Section 6.9.3
Static pre-ranking	yes	Enables pre-index time ranking of documents where hit lists are ordered first by ascending static rank, then by ascending document ID.	Section 6.9.2

Table 1.5: Zebra sorting and ranking

Feature	Availability	Notes	Reference
Incremental and batch updates		It is possible to schedule record inserts/updates/deletes in any quantity, from single individual handled records to batch updates in strikes of any size, as well as total re-indexing of all records from file system.	zebraidx(1)
Remote updates	Z39.50 extended services	Updates can be performed from remote locations using the Z39.50 extended services. Access to extended services can be login-password protected.	Section 6.10 and Section 6.2
Live updates	transaction based	Data updates are transaction based and can be performed on running Zebra systems. Full searchability is preserved during life data update due to use of shadow disk areas for update operations. Multiple update transactions at the same time are lined up, to be performed one after each other. Data integrity is preserved.	Section 6.8

Table 1.6: Zebra live updates

Feature	Availability	Notes	Reference
Fundamental operations	Z39.50/SRU `explain`, `search`, `scan`, and `update`		Section 5.1.2
Z39.50 protocol support	yes	Protocol facilities supported are: `init`, `search`, `present` (retrieval), Segmentation (support for very large records), `delete`, `scan` (index browsing), `sort`, `close` and support for the `update` Extended Service to add or replace an existing XML record. Piggy-backed presents are honored in the search request. Named result sets are supported.	the section called "Z39.50 Protocol Support and Behavior"
Web Service support	SRU	The protocol operations `explain`, `searchRetrieve` and `scan` are supported. CQL to internal query model RPN conversion is supported. Extended RPN queries for search/retrieve and scan are supported.	the section called "SRU Protocol Support and Behavior"

Table 1.7: Zebra networked protocols

Feature	Availability	Notes	Reference
No of records	40-60 million		
Data size	100 GB of record data	Zebra based applications have successfully indexed up to 100 GB of record data	
Scale out	multiple discs		
Performance	`O(n * log N)`	Zebra query speed and performance is affected roughly by `O(log N)`, where `N` is the total database size, and by `O(n)`, where `n` is the specific query hit set size.	
Average search times		Even on very large size databases hit rates of 20 queries per seconds with average query answering time of 1 second are possible, provided that the boolean queries are constructed sufficiently precise to result in hit sets of the order of 1000 to 5.000 documents.	
Large databases	64 bit file pointers	64 file pointers assure that register files can extend the 2 GB limit. Logical files can be automatically partitioned over multiple disks, thus allowing for large databases.	

Table 1.8: Zebra data size and scalability

Feature	Availability	Notes	Reference
Linux		GNU Linux (32 and 64bit), journaling Reiser or (better) JFS file system on disks. NFS file systems are not supported. GNU/Debian Linux packages are available	Section 2.2
Unix	tar-ball	Zebra is written in portable C, so it runs on most Unix-like systems. Usual tar-ball install possible on many major Unix systems	Section 2.1
Windows	NT/2000/2003/XP	Zebra runs as well on Windows (NT/2000/2003/XP). Windows installer packages available	Section 2.3

Table 1.9: Zebra supported platforms

1.3.2 Kete Open Source Digital Library and Archiving software

Kete is a digital object management repository, initially developed in New Zealand. Initial development has been a partnership between the Horowhenua Library Trust and Katipo Communications Ltd. funded as part of the Community Partnership Fund in 2006. Kete is purpose built software to enable communities to build their own digital libraries, archives and repositories.

It is based on Ruby-on-Rails and MySQL, and integrates the Zebra server and the YAZ toolkit for indexing and retrieval of it's content. Zebra is run as separate computer process from the Kete application. See how Kete manages Zebra.

Why does Kete wants to use Zebra?? Speed, Scalability and easy integration with Koha. Read their detailed reasoning here.

1.3.3 Emilda open source ILS

Emilda is a complete Integrated Library System, released under the GNU General Public License. It has a full featured Web-OPAC, allowing comprehensive system management from virtually any computer with an Internet connection, has template based layout allowing anyone to alter the visual appearance of Emilda, and is XML based language for fast and easy portability to virtually any language. Currently, Emilda is used at three schools in Espoo, Finland.

As a surplus, 100% MARC compatibility has been achieved using the Zebra Server from Index Data as backend server.

1.3.4 ReIndex.Net web based ILS

Reindex.net is a netbased library service offering all traditional functions on a very high level plus many new services. Reindex.net is a comprehensive and powerful WEB system based on standards such as XML and Z39.50. updates. Reindex supports MARC21, danMARC eller Dublin Core with UTF8-encoding.

Reindex.net runs on GNU/Debian Linux with Zebra and Simpleserver from Index Data for bibliographic data. The relational database system Sybase 9 XML is used for administrative data. Internally MARCXML is used for bibliographical records. Update utilizes Z39.50 extended services.

1.3.5 DADS - the DTV Article Database Service

DADS is a huge database of more than ten million records, totalling over ten gigabytes of data. The records are metadata about academic journal articles, primarily scientific; about 10% of these metadata records link to the full text of the articles they describe, a body of about a terabyte of information (although the full text is not indexed.)

It allows students and researchers at DTU (Danmarks Tekniske Universitet, the Technical College of Denmark) to find and order articles from multiple databases in a single query. The database contains literature on all engineering subjects. It's available on-line through a web gateway, though currently only to registered users.

More information can be found at `http://www.dtv.dk/` and `http://dads.dtv.dk`

1.3.6 Infonet Eprints

The InfoNet Eprints service from the Technical Knowledge Center of Denmark provides access to documents stored in eprint/preprint servers and institutional research archives around the world. The service is based on Open Archives Initiative metadata harvesting of selected scientific archives around the world. These open archives offer free and unrestricted access to their contents.

Infonet Eprints currently holds 1.4 million records from 16 archives. The online search facility is found at http://preprints.cvt.dk.

1.3.7 Alvis

The Alvis EU project run under the 6th Framework (IST-1-002068-STP) is building a semantic-based peer-to-peer search engine. A consortium of eleven partners from six different European Community countries plus Switzerland and China contribute with expertise in a broad range of specialties including network topologies, routing algorithms, linguistic analysis and bioinformatics.

The Zebra information retrieval indexing machine is used inside the Alvis framework to manage huge collections of natural language processed and enhanced XML data, coming from a topic relevant web crawl. In this application, Zebra swallows and manages 37GB of XML data in about 4 hours, resulting in search times of fractions of seconds.

1.3.8 ULS (Union List of Serials)

The M25 Systems Team has created a union catalogue for the periodicals of the twenty-one constituent libraries of the University of London and the University of Westminster (http://www.m25lib.ac.uk/ULS/). They have achieved this using an unusual architecture, which they describe as a ``non-distributed virtual union catalogue".

The member libraries send in data files representing their periodicals, including both brief bibliographic data and summary holdings. Then 21 individual Z39.50 targets are created, each using Zebra, and all mounted on the single hardware server. The live service provides a web gateway allowing Z39.50 searching of all of the targets or a selection of them. Zebra's small footprint allows a relatively modest system to comfortably host the 21 servers.

More information can be found at http://www.m25lib.ac.uk/ULS/

1.3.9 NLI-Z39.50 - a Natural Language Interface for Libraries

Fernuniversität Hagen in Germany have developed a natural language interface for access to library databases. In order to evaluate this interface for recall and precision, they chose Zebra as the basis for retrieval effectiveness. The Zebra server contains a copy of the GIRT database, consisting of more than 76000 records in SGML format (bibliographic records from social science), which are mapped to MARC for presentation.

(GIRT is the German Indexing and Retrieval Testdatabase. It is a standard German-language test database for intelligent indexing and retrieval systems. See http://www.gesis.org/forschung/informations clef-delos.htm)

Evaluation will take place as part of the TREC/CLEF campaign 2003 http://clef.iei.pi.cnr.it.

For more information, contact Johannes Leveling Johannes.Leveling@FernUni-Hagen.De

1.3.10 Various web indexes

Zebra has been used by a variety of institutions to construct indexes of large web sites, typically in the region of tens of millions of pages. In this role, it functions somewhat similarly to the engine of Google or AltaVista, but for a selected intranet or a subset of the whole Web.

For example, Liverpool University's web-search facility (see on the home page at http://www.liv.ac.uk/ and many sub-pages) works by relevance-searching a Zebra database which is populated by the Harvest-NG web-crawling software.

For more information on Liverpool university's intranet search architecture, contact John Gilbertson jgilbert@liver

Kang-Jin Lee has recently modified the Harvest web indexer to use Zebra as its native repository engine. His comments on the switch over from the old engine are revealing:

> The first results after some testing with Zebra are very promising. The tests were done with around 220,000 SOIF files, which occupies 1.6GB of disk space.
>
> Building the index from scratch takes around one hour with Zebra where [old-engine] needs around five hours. While [old-engine] blocks search requests when updating its index, Zebra can still answer search requests. [...] Zebra supports incremental indexing which will speed up indexing even further.
>
> While the search time of [old-engine] varies from some seconds to some minutes depending how expensive the query is, Zebra usually takes around one to three seconds, even for expensive queries. [...] Zebra can search more than 100 times faster than [old-engine] and can process multiple search requests simultaneously
>
> I am very happy to see such nice software available under GPL.

1.4 Support

You can get support for Zebra from at least three sources.

First, there's the Zebra web site at http://www.indexdata.com/zebra/, which always has the most recent version available for download. If you have a problem with Zebra, the first thing to do is see whether it's fixed in the current release.

Second, there's the Zebra mailing list. Its home page at http://lists.indexdata.dk/cgi-bin/mailman/listinfo/zebralist includes a complete archive of all messages that have ever been posted on the list. The Zebra mailing list is used both for announcements from the authors (new releases, bug fixes, etc.) and general discussion. You are welcome to seek support there. Join by filling the form on the list home page.

Third, it's possible to buy a commercial support contract, with well defined service levels and response times, from Index Data. See http://www.indexdata.com/support/ for details.

Chapter 2

Installation

Zebra is written in ANSI C and was implemented with portability in mind. We primarily use GCC on UNIX and Microsoft Visual C++ on Windows.

The software is regularly tested on Debian GNU/Linux, Red Hat Linux, Gentoo Linux, SuSE Linux, FreeBSD (i386), MAC OSX, SunOS 5.9 (sparc), Windows 2000.

Zebra can be configured to use the following utilities (most of which are optional):

YAZ (required) Zebra uses YAZ to support Z39.50 / SRU. Zebra also uses a lot of other utilities (not related to networking), such as memory management and XML support.

For the DOM XML / ALVIS record filters, YAZ must be compiled with Libxml2 and Libxslt support and Libxml2 must be version 2.6.15 or later.

iconv (optional) Character set conversion. This is required if you're going to use any other character set than UTF-8 and ISO-8859-1 for records. Note that some Unixes has iconv built-in.

Expat (optional) XML parser. If you're going to index real XML you should install this (filter grs.xml). On most systems you should be able to find binary Expat packages.

Tcl (optional) Tcl is required if you need to use the Tcl record filter for Zebra. You can find binary packages for Tcl for many Unices and Windows.

Autoconf, Automake (optional) GNU Automake and Autoconf are only required if you're using the CVS version of Zebra. You do not need these if you have fetched a Zebra tar.

Docbook and friends (optional) These tools are only required if you're writing documentation for Zebra. You need the following Debian packages: docbook, docbook-xml, docbook-xsl, docbook-utils, xsltproc.

2.1 UNIX

On Unix, GCC works fine, but any native C compiler should be possible to use as long as it is ANSI C compliant.

Unpack the distribution archive. The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile` in each directory of Zebra.

To run the configure script type:

```
./configure
```

The configure script attempts to use C compiler specified by the `CC` environment variable. If this is not set, `cc` or GNU C will be used. The `CFLAGS` environment variable holds options to be passed to the C compiler. If you're using a Bourne-shell compatible shell you may pass something like this:

```
CC=/opt/ccs/bin/cc CFLAGS=-O ./configure
```

The configure script support various options: you can see what they are with

```
./configure --help
```

Once the build environment is configured, build the software by typing:

```
make
```

If the build is successful, two executables are created in the sub-directory `index`:

`zebrasrv` The Z39.50 server and search engine.

`zebraidx` The administrative indexing tool.

`index/*.so` The `.so`-files are Zebra record filter modules. There are modules for reading MARC (`mod-grs-marc.so`), XML (`mod-grs-xml.so`) , etc.

Note

Using configure option `--disable-shared` builds Zebra statically and links "in" Zebra filter code statically, i.e. no `.so-files` are generated

---

You can now use Zebra. If you wish to install it system-wide, then as root type

```
make install
```

By default this will install the Zebra executables in `/usr/local/bin`, and the standard configuration files in `/usr/local/share/idzebra-2.0`. If shared modules are built, these are installed in `/usr/local/lib/idzebra-2.0/modules`. You can override this with the `--prefix` option to configure.

2.2 GNU/Debian

2.2.1 GNU/Debian Linux on i686 Platform

Index Data provides pre-compiled GNU/Debian i686 Linux packages at our Debian package archive, both for the Sarge and the Etch release.

To install these packages, you need to add two lines to your `/etc/apt/sources.list` configuration file, either the Sarge sources found at

```
deb http://ftp.indexdata.dk/debian sarge main
deb-src http://ftp.indexdata.dk/debian sarge main
```

or the Etch sources from

```
deb http://ftp.indexdata.dk/debian etch main
deb-src http://ftp.indexdata.dk/debian etch main
```

After refreshing the package cache with the command

```
apt-get update
```

as `root`, the Zebra indexer is easily installed issuing

```
apt-get install idzebra-2.0 idzebra-2.0-doc
```

2.2.2 Ubuntu/Debian and GNU/Debian on other platforms

These Zebra packages are specifically compiled for GNU/Debian Linux systems. Installation on other GNU/Debian systems is possible by re-compilation the Debian way: you need to add only the `deb-src` sources lines to the `/etc/apt/sources.list` configuration file, that is either the Sarge sources

```
deb-src http://ftp.indexdata.dk/debian sarge main
```

or the Etch sources

```
deb-src http://ftp.indexdata.dk/debian etch main
```

After refreshing the package cache with the command

```
apt-get update
apt-get build-dep idzebra-2.0
```

as `root`, the Zebra indexer is recompiled and installed issuing

```
fakeroot apt-get source --compile idzebra-2.0
```

as normal user. The compiled GNU/Debian packages can then be installed as `root` issuing

```
dpkg -i install idzebra-2.0*.deb libidzebra-2.0*.deb
```

2.3 WIN32

The easiest way to install Zebra on Windows is by downloading an installer from here. The installer comes with source too - in case you wish to compile Zebra with different Compiler options.

Zebra is shipped with "makefiles" for the NMAKE tool that comes with Microsoft Visual C++. Version 2003 and 2005 has been tested. We expect that zebra compiles with version 6 as well.

Start a command prompt and switch the sub directory `WIN` where the file `makefile` is located. Customize the installation by editing the `makefile` file (for example by using notepad). The following summarizes the most important settings in that file:

DEBUG If set to 1, the software is compiled with debugging libraries (code generation is multi-threaded debug DLL). If set to 0, the software is compiled with release libraries (code generation is multi-threaded DLL).

YAZDIR Directory of YAZ source. Zebra's makefile expects to find `yaz.lib`, `yaz.dll` in *yazdir*`/lib` and *yazdir*`/bin` respectively.

HAVE_EXPAT, EXPAT_DIR If `HAVE_EXPAT` is set to 1, Zebra is compiled with Expat support. In this configuration, set `ZEBRA_DIR` to the Expat source directory. Windows version of Expat can be downloaded from SourceForge.

HAVE_ICONV, ICONV_DIR If `HAVE_ICONV` is set to 1, Zebra is compiled with iconv support. In this configuration, set `ICONV_DIR` to the iconv source directory. Iconv binaries can be downloaded from this site.

BZIP2INCLUDE, BZIP2LIB, BZIP2DEF Define these symbols if Zebra is to be compiled with BZIP2 record compression support.

Warning
The `DEBUG` setting in the makefile for Zebra must be set to the same value as `DEBUG` setting in the makefile for YAZ. If not, the Zebra server/indexer will crash.

When satisfied with the settings in the makefile, type

```
nmake
```

Note
If the `nmake` command is not found on your system you probably haven't defined the environment variables required to use that tool. To fix that, find and run the batch file `vcvars32.bat`. You need to run it from within the command prompt or set the environment variables "globally"; otherwise it doesn't work.

If you wish to recompile Zebra - for example if you modify settings in the `makefile` you can delete object files, etc by running.

```
nmake clean
```

The following files are generated upon successful compilation:

bin/zebraidx.exe The Zebra indexer.

bin/zebrasrv.exe The Zebra server.

2.4 Upgrading from Zebra version 1.3.x

Zebra's installation directories have changed a bit. In addition, the new loadable modules must be defined in the master `zebra.cfg` configuration file. The old version 1.3.x configuration options

```
# profilePath - where to look for config files
profilePath: some/local/path:/usr/share/idzebra/tab
```

must be changed to

```
# profilePath - where to look for config files
profilePath: some/local/path:/usr/share/idzebra-2.0/tab

# modulePath - where to look for loadable zebra modules
modulePath: /usr/lib/idzebra-2.0/modules
```

Note
The internal binary register structures have changed; all Zebra databases must be re-indexed after upgrade.

The attribute set definition files may no longer contain redirection to other fields. For example the following snippet of a custom `custom/bib1.att` BIB-1 attribute set definition file is no longer supported:

```
att 1016            Any    1016,4,1005,62
```

and should be changed to

```
att 1016            Any
```

Similar behaviour can be expressed in the new release by defining a new index `Any:w` in all GRS-1 `*.abs` record indexing configuration files. The above example configuration needs to make the changes from version 1.3.x indexing instructions

```
xelm /*/alternative  Body-of-text:w,Title:s,Title:w
xelm /*/title        Body-of-text:w,Title:s,Title:w
```

to version 2.0.0 indexing instructions

```
xelm /*/alternative  Any:w,Body-of-text:w,Title:s,Title:w
xelm /*/title        Any:w,Body-of-text:w,Title:s,Title:w
```

It is also possible to map the numerical attribute value `@attr 1=1016` onto another already existing huge index, in this example, one could for example use the mapping

```
att 1016            Body-of-text
```

with equivalent outcome without editing all GRS-1 `*.abs` record indexing configuration files.

Server installations which use the special `IDXPATH` attribute set must add the following line to the `zebra.cfg` configuration file:

```
attset: idxpath.att
```

Chapter 3

Tutorial

3.1 A first OAI indexing example

In this section, we will test the system by indexing a small set of sample OAI records that are included with the Zebra distribution, running a Zebra server against the newly created database, and searching the indexes with a client that connects to that server.

Go to the `examples/oai-pmh` subdirectory of the distribution archive, or make a deep copy of the Debian installation directory `/usr/share/idzebra-2.0.-examples/oai-pmh`. An XML file containing multiple OAI records is located in the sub directory `examples/oai-pmh/data`.

Additional OAI test records can be downloaded by running a shell script (you may want to abort the script when you have waited longer than your coffee brews ..).

```
cd data
./fetch_OAI_data.sh
cd ../
```

To index these OAI records, type:

```
zebraidx-2.0 -c conf/zebra.cfg init
zebraidx-2.0 -c conf/zebra.cfg update data
zebraidx-2.0 -c conf/zebra.cfg commit
```

In case you have not installed zebra yet but have compiled the binaries from this tarball, use the following command form:

```
../../index/zebraidx -c conf/zebra.cfg this and that
```

On some systems the Zebra binaries are installed under the generic names, you need to use the following command form:

```
zebraidx -c conf/zebra.cfg this and that
```

In this command, the word `update` is followed by the name of a directory: `zebraidx` updates all files in the hierarchy rooted at `data`. The command option `-c conf/zebra.cfg` points to the proper configuration file.

You might ask yourself how XML content is indexed using XSLT stylesheets: to satisfy your curiosity, you might want to run the indexing transformation on an example debugging OAI record.

```
xsltproc conf/oai2index.xsl data/debug-record.xml
```

Here you see the OAI record transformed into the indexing XML format. Zebra is creating several inverted indexes, and their name and type are clearly visible in the indexing XML format.

If your indexing command was successful, you are now ready to fire up a server. To start a server on port 9999, type:

```
zebrasrv-2.0 -c conf/zebra.cfg  @:9999
```

The Zebra index that you have just created has a single database named `Default`. The database contains several OAI records, and the server will return records in the XML format only. The indexing machine did the splitting into individual records just behind the scenes.

3.2 Searching the OAI database by web service

Zebra has a build-in web service, which is close to the SRU standard web service. We use it to access our new database using any XML enabled web browser. This service is using the PQF query language. In a later section we show how to run a fully compliant SRU server, including support for the query language CQL

Searching and retrieving XML records is easy. For example, you can point your browser to one of the following URLs to search for the term `the`. Just point your browser at this link: http://localhost:9999/-?version=1.1&operation=searchRetrieve&x-pquery=the

Warning
These URLs won't work unless you have indexed the example data and started an Zebra server as outlined in the previous section.

In case we actually want to retrieve one record, we need to alter our URL to the following http://localhost:9999/-?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRecords=1&recordSchema=d

This way we can page through our result set in chunks of records, for example, we access the 6th to the 10th record using the URL http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=

3.3 Presenting search results in different formats

Zebra uses XSLT stylesheets for both XMLrecord indexing and display retrieval. In this example installation, they are two retrieval schema's defined in `conf/dom-conf.xml`: the `dc` schema implemented in `conf/oai2dc.xsl`, and the `zebra` schema implemented in `conf/oai2zebra.xsl`. The

URLs for accessing both are the same, except for the different value of the `recordSchema` parameter: http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRecord and http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRe For the curious, one can see that the XSLT transformations really do the magic.

```
xsltproc conf/oai2dc.xsl data/debug-record.xml
xsltproc conf/oai2zebra.xsl data/debug-record.xml
```

Notice also that the Zebra specific parameters are injected by the engine when retrieving data, therefore some of the attributes in the `zebra` retrieval schema are not filled when running the transformation from the command line.

In addition to the user defined retrieval schema's one can always choose from many build-in schema's. In case one is only interested in the Zebra internal metadata about a certain record, one uses the `zebra::meta` schema. http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maxi

The `zebra::data` schema is used to retrieve the original stored OAI XML record. http://localhost:9999/-?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRecords=1&recordSchema=

3.4 More interesting searches

The OAI indexing example defines many different index names, a study of the `conf/oai2index.xsl` stylesheet reveals the following word type indexes (i.e. those with suffix `:w`):

```
any:w
title:w
author:w
subject:w
description:w
contributor:w
publisher:w
language:w
rights:w
```

By default, searches do access the `any:w` index, but we can direct searches to any access point by constructing the correct PQF query. For example, to search in titles only, we use http://localhost:9999/-?version=1.1&operation=searchRetrieve&x-pquery=@attr 1=title the&startRecord=1&maximumRecords=1&rec

Similar we can direct searches to the other indexes defined. Or we can create boolean combinations of searches on different indexes. In this case we search for `the` in `title` and for `fish` in `description` using the query `@and @attr 1=title the @attr 1=description fish`. http://localhost:9999/-?version=1.1&operation=searchRetrieve&x-pquery=@and @attr 1=title the @attr 1=description fish&startRecord

3.5 Investigating the content of the indexes

How does the magic work? What is inside the indexes? Why is a certain record found by a search, and another not?. The answer is in the inverted indexes. You can easily investigate them using the special Zebra schema `zebra::index::fieldname`. In this example you can see that the `title` index has both word (type `:w`) and phrase (type `:p`) indexed fields, http://localhost:9999/?version=1.1&operation=searchRetrieve&
pquery=the&startRecord=1&maximumRecords=1&recordSchema=zebra::index::title

But where in the indexes did the term match for the query occur? Easily answered with the special Zebra schema `zebra::snippet`. The matching terms are encapsulated by `<s>` tags. http://localhost:9999/-
?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRecords=1&recordSchema=z

How can I refine my search? Which interesting search terms are found inside my hit set? Try the special Zebra schema `zebra::facet::fieldname:type`. In this case, we investigate additional search terms for the `title:w` index. http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery=the&startReco

One can ask for multiple facets. Here, we want them from phrase indexes of type `:p`. http://localhost:9999/-
?version=1.1&operation=searchRetrieve&x-pquery=the&startRecord=1&maximumRecords=1&recordSchema=z

3.6 Setting up a correct SRU web service

The SRU specification mandates that the CQL query language is supported and properly configure. Also, the server needs to be able to emit a proper Explain XML record, which is used to determine the capabilities of the specific server instance.

In this example configuration we exploit the similarities between the Explain record and the CQL query language configuration, we generate the later from the former using an XSLT transformation.

```
xsltproc conf/explain2cqlpqftxt.xsl conf/explain.xml > conf/cql2pqf. ↩
   txt
```

We are all set to start the SRU/acro.z3950; server including PQF and CQL query configuration. It uses the YAZ frontend server configuration - just type

```
zebrasrv -f conf/yazserver.xml
```

First, we'd like to be sure that we can see the Explain XML response correctly. You might use either of these equivalent requests: http://localhost:9999 http://localhost:9999/?version=1.1&operation=explain

Now we can issue true SRU requests. For example, `dc.title=the and dc.description=fish` results in the following page http://localhost:9999/?version=1.1&operation=searchRetrieve&query=dc.title=the
and dc.description=fish &startRecord=1&maximumRecords=1&recordSchema=dc

Scan of indexes is a part of the SRU server business. For example, scanning the `dc.title` index gives us an idea what search terms are found there http://localhost:9999/?version=1.1&operation=scan&scanClause=dc.title
, whereas http://localhost:9999/?version=1.1&operation=scan&scanClause=dc.identifier=fish accesses the indexed identifiers.

In addition, all Zebra internal special element sets or record schema's of the form `zebra::` just work right out of the box http://localhost:9999/?version=1.1&operation=searchRetrieve&query=dc.title=the and dc.description=fish &startRecord=1&maximumRecords=1&recordSchema=zebra::snippet

3.7 Searching the OAI database by Z39.50 protocol

In this section we repeat the searches and presents we have done so far using the binary Z39.50 protocol, you can use any Z39.50 client. For instance, you can use the demo command-line client that comes with YAZ.

Connecting to the server is done by the command

```
yaz-client localhost:9999
```

When the client has connected, you can type:

```
Z> format xml
Z> querytype prefix
Z> elements oai
Z> find the
Z> show 1+1
```

Z39.50 presents using presentation stylesheets:

```
Z> elements dc
Z> show 2+1

Z> elements zebra
Z> show 3+1
```

Z39.50 buildin Zebra presents (in this configuration only if started without yaz-frontendserver):

```
Z> elements zebra::meta
Z> show 4+1

Z> elements zebra::meta::sysno
Z> show 5+1

Z> format sutrs
Z> show 5+1
Z> format xml

Z> elements zebra::index
Z> show 6+1

Z> elements zebra::snippet
Z> show 7+1

Z> elements zebra::facet::any:w
Z> show 1+1
```

```
Z> elements zebra::facet::publisher:p,title:p
Z> show 1+1
```

Z39.50 searches targeted at specific indexes and boolean combinations of these can be issued as well.

```
Z> elements dc
Z> find @attr 1=oai_identifier @attr 4=3 oai:caltechcstr.library. ↩
   caltech.edu:4
Z> show 1+1

Z> find @attr 1=oai_datestamp @attr 4=3 2001-04-20
Z> show 1+1

Z> find @attr 1=oai_setspec @attr 4=3 7374617475733D756E707562
Z> show 1+1

Z> find @attr 1=title communication
Z> show 1+1

Z> find @attr 1=identifier @attr 4=3
http://resolver.caltech.edu/CaltechCSTR:1986.5228-tr-86
Z> show 1+1
```

etc, etc.

Z39.50 scan:

```
yaz-client localhost:9999
Z> format xml
Z> querytype prefix
Z> scan @attr 1=oai_identifier @attr 4=3 oai
Z> scan @attr 1=oai_datestamp @attr 4=3 1
Z> scan @attr 1=oai_setspec @attr 4=3 2000
Z>
Z> scan @attr 1=title communication
Z> scan @attr 1=identifier @attr 4=3 a
```

Z39.50 search using server-side CQL conversion:

```
Z> format xml
Z> querytype cql
Z> elements dc
Z>
Z> find harry
Z>
Z> find dc.creator = the
Z> find dc.creator = the
Z> find dc.title = the
Z>
Z> find dc.description < the
Z> find dc.title > some
```

```
Z>
Z> find dc.identifier="http://resolver.caltech.edu/CaltechCSTR ↩
   :1978.2276-tr-78"
Z> find dc.relation = something
```

Tip
Z39.50 scan using server side CQL conversion - unfortunately, this will _never_ work as it is not supported by the Z39.50 standard. If you want to use scan using server side CQL conversion, you need to make an SRW connection using yaz-client, or a SRU connection using REST Web Services - any browser will do.

Tip
All indexes defined by 'type="0"' in the indexing style sheet must be searched using the '@attr 4=3' structure attribute instruction.

Notice that searching and scan on indexes `contributor`, `language`, `rights`, and `source` might fail, simply because none of the records in the small example set have these fields set, and consequently, these indexes might not been created.

Chapter 4

Overview of Zebra Architecture

4.1 Local Representation

As mentioned earlier, Zebra places few restrictions on the type of data that you can index and manage. Generally, whatever the form of the data, it is parsed by an input filter specific to that format, and turned into an internal structure that Zebra knows how to handle. This process takes place whenever the record is accessed - for indexing and retrieval.

The RecordType parameter in the `zebra.cfg` file, or the `-t` option to the indexer tells Zebra how to process input records. Two basic types of processing are available - raw text and structured data. Raw text is just that, and it is selected by providing the argument *text* to Zebra. Structured records are all handled internally using the basic mechanisms described in the subsequent sections. Zebra can read structured records in many different formats.

4.2 Main Components

The Zebra system is designed to support a wide range of data management applications. The system can be configured to handle virtually any kind of structured data. Each record in the system is associated with a *record schema* which lends context to the data elements of the record. Any number of record schemas can coexist in the system. Although it may be wise to use only a single schema within one database, the system poses no such restrictions.

The Zebra indexer and information retrieval server consists of the following main applications: the **zebraidx** indexing maintenance utility, and the **zebrasrv** information query and retrieval server. Both are using some of the same main components, which are presented here.

The virtual Debian package `idzebra-2.0` installs all the necessary packages to start working with Zebra - including utility programs, development libraries, documentation and modules.

4.2.1 Core Zebra Libraries Containing Common Functionality

The core Zebra module is the meat of the **zebraidx** indexing maintenance utility, and the **zebrasrv** information query and retrieval server binaries. Shortly, the core libraries are responsible for

Dynamic Loading of external filter modules, in case the application is not compiled statically. These filter modules define indexing, search and retrieval capabilities of the various input formats.

Index Maintenance Zebra maintains Term Dictionaries and ISAM index entries in inverted index structures kept on disk. These are optimized for fast inset, update and delete, as well as good search performance.

Search Evaluation by execution of search requests expressed in PQF/RPN data structures, which are handed over from the YAZ server frontend API. Search evaluation includes construction of hit lists according to boolean combinations of simpler searches. Fast performance is achieved by careful use of index structures, and by evaluation specific index hit lists in correct order.

Ranking and Sorting components call resorting/re-ranking algorithms on the hit sets. These might also be pre-sorted not only using the assigned document ID's, but also using assigned static rank information.

Record Presentation returns - possibly ranked - result sets, hit numbers, and the like internal data to the YAZ server backend API for shipping to the client. Each individual filter module implements it's own specific presentation formats.

The Debian package `libidzebra-2.0` contains all run-time libraries for Zebra, the documentation in PDF and HTML is found in `idzebra-2.0-doc`, and `idzebra-2.0-common` includes common essential Zebra configuration files.

4.2.2 Zebra Indexer

The **zebraidx** indexing maintenance utility loads external filter modules used for indexing data records of different type, and creates, updates and drops databases and indexes according to the rules defined in the filter modules.

The Debian package `idzebra-2.0-utils` contains the **zebraidx** utility.

4.2.3 Zebra Searcher/Retriever

This is the executable which runs the Z39.50/SRU/SRW server and glues together the core libraries and the filter modules to one great Information Retrieval server application.

The Debian package `idzebra-2.0-utils` contains the **zebrasrv** utility.

4.2.4 YAZ Server Frontend

The YAZ server frontend is a full fledged stateful Z39.50 server taking client connections, and forwarding search and scan requests to the Zebra core indexer.

In addition to Z39.50 requests, the YAZ server frontend acts as HTTP server, honoring SRU SOAP requests, and SRU REST requests. Moreover, it can translate incoming CQL queries to PQF queries, if correctly configured.

YAZ is an Open Source toolkit that allows you to develop software using the ANSI Z39.50/ISO23950 standard for information retrieval. It is packaged in the Debian packages `yaz` and `libyaz`.

4.2.5 Record Models and Filter Modules

The hard work of knowing *what* to index, *how* to do it, and *which* part of the records to send in a search/retrieve response is implemented in various filter modules. It is their responsibility to define the exact indexing and record display filtering rules.

The virtual Debian package `libidzebra-2.0-modules` installs all base filter modules.

4.2.5.1 DOM XML Record Model and Filter Module

The DOM XML filter uses a standard DOM XML structure as internal data model, and can thus parse, index, and display any XML document.

A parser for binary MARC records based on the ISO2709 library standard is provided, it transforms these to the internal MARCXML DOM representation.

The internal DOM XML representation can be fed into four different pipelines, consisting of arbitrarily many successive XSLT transformations; these are for

- input parsing and initial transformations,
- indexing term extraction transformations
- transformations before internal document storage, and
- retrieve transformations from storage to output format

The DOM XML filter pipelines use XSLT (and if supported on your platform, even EXSLT), it brings thus full XPATH support to the indexing, storage and display rules of not only XML documents, but also binary MARC records.

Finally, the DOM XML filter allows for static ranking at index time, and to to sort hit lists according to predefined static ranks.

Details on the experimental DOM XML filter are found in Chapter 7.

The Debian package `libidzebra-2.0-mod-dom` contains the DOM filter module.

4.2.5.2 ALVIS XML Record Model and Filter Module

Note
The functionality of this record model has been improved and replaced by the DOM XML record model. See Section 4.2.5.1.

The Alvis filter for XML files is an XSLT based input filter. It indexes element and attribute content of any thinkable XML format using full XPATH support, a feature which the standard Zebra GRS-1 SGML and XML filters lacked. The indexed documents are parsed into a standard XML DOM tree, which restricts record size according to availability of memory.

The Alvis filter uses XSLT display stylesheets, which let the Zebra DB administrator associate multiple, different views on the same XML document type. These views are chosen on-the-fly in search time.

In addition, the Alvis filter configuration is not bound to the arcane BIB-1 Z39.50 library catalogue indexing traditions and folklore, and is therefore easier to understand.

Finally, the Alvis filter allows for static ranking at index time, and to to sort hit lists according to predefined static ranks. This imposes no overhead at all, both search and indexing perform still *O(1)* irrespectively of document collection size. This feature resembles Google's pre-ranking using their PageRank algorithm.

Details on the experimental Alvis XSLT filter are found in Chapter 8.

The Debian package `libidzebra-2.0-mod-alvis` contains the Alvis filter module.

4.2.5.3 GRS-1 Record Model and Filter Modules

Note
The functionality of this record model has been improved and replaced by the DOM XML record model. See Section 4.2.5.1.

The GRS-1 filter modules described in Chapter 9 are all based on the Z39.50 specifications, and it is absolutely mandatory to have the reference pages on BIB-1 attribute sets on you hand when configuring GRS-1 filters. The GRS filters come in different flavors, and a short introduction is needed here. GRS-1 filters of various kind have also been called ABS filters due to the `*.abs` configuration file suffix.

The *grs.marc* and *grs.marcxml* filters are suited to parse and index binary and XML versions of traditional library MARC records based on the ISO2709 standard. The Debian package for both filters is `libidzebra-2.0-mod-grs-marc`.

GRS-1 TCL scriptable filters for extensive user configuration come in two flavors: a regular expression filter *grs.regx* using TCL regular expressions, and a general scriptable TCL filter called *grs.tcl* are both included in the `libidzebra-2.0-mod-grs-regx` Debian package.

A general purpose SGML filter is called *grs.sgml*. This filter is not yet packaged, but planned to be in the `libidzebra-2.0-mod-grs-sgml` Debian package.

The Debian package `libidzebra-2.0-mod-grs-xml` includes the *grs.xml* filter which uses Expat to parse records in XML and turn them into IDZebra's internal GRS-1 node trees. Have also a look at the Alvis XML/XSLT filter described in the next session.

4.2.5.4 TEXT Record Model and Filter Module

Plain ASCII text filter. TODO: add information here.

4.3 Indexing and Retrieval Workflow

Records pass through three different states during processing in the system.

- When records are accessed by the system, they are represented in their local, or native format. This might be SGML or HTML files, News or Mail archives, MARC records. If the system doesn't already know how to read the type of data you need to store, you can set up an input filter by preparing conversion rules based on regular expressions and possibly augmented by a flexible scripting language (Tcl). The input filter produces as output an internal representation, a tree structure.
- When records are processed by the system, they are represented in a tree-structure, constructed by tagged data elements hanging off a root node. The tagged elements may contain data or yet more tagged elements in a recursive structure. The system performs various actions on this tree structure (indexing, element selection, schema mapping, etc.),
- Before transmitting records to the client, they are first converted from the internal structure to a form suitable for exchange over the network - according to the Z39.50 standard.

4.4 Retrieval of Zebra internal record data

Starting with `Zebra` version 2.0.5 or newer, it is possible to use a special element set which has the prefix `zebra::`.

Using this element will, regardless of record type, return Zebra's internal index structure/data for a record. In particular, the regular record filters are not invoked when these are in use. This can in some cases make the retrieval faster than regular retrieval operations (for MARC, XML etc).

Element Set	**Description**
`zebra::meta::sysno`	Get Zebra record system ID
`zebra::data`	Get raw record
`zebra::meta`	Get Zebra record internal metadata
`zebra::index`	Get all indexed keys for record
`zebra::index::`*`f`*	Get indexed keys for field *`f`* for record
`zebra::index::`*`f`*`:`*`t`*	Get indexed keys for field *`f`* and type *`t`* for record
`zebra::snippet`	Get snippet for record for one or more indexes (f1,f2,..). This includes a phrase from the original record at the point where a match occurs (for a query). By default give terms before - and after are included in the snippet. The matching terms are enclosed within element `<s>`. The snippet facility requires Zebra 2.0.16 or later.
`zebra::facet::`*`f1`*`:`*`t1`*`,`*`f2`*`:`*`t2`*`,..`	Get facet of a result set. The facet result is returned as if it was a normal record, while in reality is a recap of most "important" terms in a result set for the fields given. The facet facility first appeared in Zebra 2.0.20.

Table 4.1: Special Retrieval Elements

For example, to fetch the raw binary record data stored in the zebra internal storage, or on the filesystem, the following commands can be issued:

```
Z> f @attr 1=title my
Z> format xml
Z> elements zebra::data
Z> s 1+1
Z> format sutrs
Z> s 1+1
Z> format usmarc
Z> s 1+1
```

The special `zebra::data` element set name is defined for any record syntax, but will always fetch the raw record data in exactly the original form. No record syntax specific transformations will be applied to the raw record data.

Also, Zebra internal metadata about the record can be accessed:

```
Z> f @attr 1=title my
Z> format xml
Z> elements zebra::meta::sysno
Z> s 1+1
```

displays in `XML` record syntax only internal record system number, whereas

```
Z> f @attr 1=title my
Z> format xml
Z> elements zebra::meta
Z> s 1+1
```

displays all available metadata on the record. These include system number, database name, indexed filename, filter used for indexing, score and static ranking information and finally bytesize of record.

Sometimes, it is very hard to figure out what exactly has been indexed how and in which indexes. Using the indexing stylesheet of the Alvis filter, one can at least see which portion of the record went into which index, but a similar aid does not exist for all other indexing filters.

The special `zebra::index` element set names are provided to access information on per record indexed fields. For example, the queries

```
Z> f @attr 1=title my
Z> format sutrs
Z> elements zebra::index
Z> s 1+1
```

will display all indexed tokens from all indexed fields of the first record, and it will display in `SUTRS` record syntax, whereas

```
Z> f @attr 1=title my
Z> format xml
Z> elements zebra::index::title
Z> s 1+1
Z> elements zebra::index::title:p
Z> s 1+1
```

displays in XML record syntax only the content of the zebra string index title, or even only the type p phrase indexed part of it.

Note
Trying to access numeric BIB-1 use attributes or trying to access non-existent zebra intern string access points will result in a Diagnostic 25: Specified element set 'name not valid for specified database.

Chapter 5

Query Model

5.1 Query Model Overview

5.1.1 Query Languages

Zebra is born as a networking Information Retrieval engine adhering to the international standards Z39.50 and SRU, and implement the type-1 Reverse Polish Notation (RPN) query model defined there. Unfortunately, this model has only defined a binary encoded representation, which is used as transport packaging in the Z39.50 protocol layer. This representation is not human readable, nor defines any convenient way to specify queries.

Since the type-1 (RPN) query structure has no direct, useful string representation, every client application needs to provide some form of mapping from a local query notation or representation to it.

5.1.1.1 Prefix Query Format (PQF)

Index Data has defined a textual representation in the Prefix Query Format, short *PQF*, which maps one-to-one to binary encoded *type-1 RPN* queries. PQF has been adopted by other parties developing Z39.50 software, and is often referred to as *Prefix Query Notation*, or in short PQN. See Section 5.2 for further explanations and descriptions of Zebra's capabilities.

5.1.1.2 Common Query Language (CQL)

The query model of the type-1 RPN, expressed in PQF/PQN is natively supported. On the other hand, the default SRU web services *Common Query Language* CQL is not natively supported.

Zebra can be configured to understand and map CQL to PQF. See Section 5.4.

5.1.2 Operation types

Zebra supports all of the three different Z39.50/SRU operations defined in the standards: explain, search, and scan. A short description of the functionality and purpose of each is quite in order here.

5.1.2.1 Explain Operation

The *syntax* of Z39.50/SRU queries is well known to any client, but the specific *semantics* - taking into account a particular servers functionalities and abilities - must be discovered from case to case. Enters the explain operation, which provides the means for learning which *fields* (also called *indexes* or *access points*) are provided, which default parameter the server uses, which retrieve document formats are defined, and which specific parts of the general query model are supported.

The Z39.50 embeds the explain operation by performing a search in the magic `IR-Explain-1` database; see Section 5.2.2.

In SRU, explain is an entirely separate operation, which returns an ZeeRex XML record according to the structure defined by the protocol.

In both cases, the information gathered through explain operations can be used to auto-configure a client user interface to the servers capabilities.

5.1.2.2 Search Operation

Search and retrieve interactions are the raison d'être. They are used to query the remote database and return search result documents. Search queries span from simple free text searches to nested complex boolean queries, targeting specific indexes, and possibly enhanced with many query semantic specifications. Search interactions are the heart and soul of Z39.50/SRU servers.

5.1.2.3 Scan Operation

The scan operation is a helper functionality, which operates on one index or access point a time.

It provides the means to investigate the content of specific indexes. Scanning an index returns a handful of terms actually found in the indexes, and in addition the scan operation returns the number of documents indexed by each term. A search client can use this information to propose proper spelling of search terms, to auto-fill search boxes, or to display controlled vocabularies.

5.2 RPN queries and semantics

The PQF grammar is documented in the YAZ manual, and shall not be repeated here. This textual PQF representation is not transmitted to Zebra during search, but it is in the client mapped to the equivalent Z39.50 binary query parse tree.

5.2.1 RPN tree structure

The RPN parse tree - or the equivalent textual representation in PQF - may start with one specification of the *attribute set* used. Following is a query tree, which consists of *atomic query parts (APT)* or *named result sets*, eventually paired by *boolean binary operators*, and finally *recursively combined* into complex query trees.

5.2.1.1 Attribute sets

Attribute sets define the exact meaning and semantics of queries issued. Zebra comes with some predefined attribute set definitions, others can easily be defined and added to the configuration.

Attribute set	PQF notation (Short hand)	Status	Notes
Explain	`exp-1`	Special attribute set used on the special automagic `IR-Explain-1` database to gain information on server capabilities, database names, and database and semantics.	predefined
BIB-1	`bib-1`	Standard PQF query language attribute set which defines the semantics of Z39.50 searching. In addition, all of the non-use attributes (types 2-14) define the hard-wired Zebra internal query processing.	default
GILS	`gils`	Extension to the BIB-1 attribute set.	predefined

Table 5.1: Attribute sets predefined in Zebra

The use attributes (type 1) mappings the predefined attribute sets are found in the attribute set configuration files `tab/*.att`.

Note
The Zebra internal query processing is modeled after the BIB-1 attribute set, and the non-use attributes type 2-6 are hard-wired in. It is therefore essential to be familiar with Section 5.2.4.

5.2.1.2 Boolean operators

A pair of sub query trees, or of atomic queries, is combined using the standard boolean operators into new query trees. Thus, boolean operators are always internal nodes in the query tree.

For example, we can combine the terms *information* and *retrieval* into different searches in the default index of the default attribute set as follows. Querying for the union of all documents containing the terms *information* OR *retrieval*:

Keyword	Operator	Description
`@and`	binary AND operator	Set intersection of two atomic queries hit sets
`@or`	binary OR operator	Set union of two atomic queries hit sets
`@not`	binary AND NOT operator	Set complement of two atomic queries hit sets
`@prox`	binary PROXIMITY operator	Set intersection of two atomic queries hit sets. In addition, the intersection set is purged for all documents which do not satisfy the requested query term proximity. Usually a proper subset of the AND operation.

Table 5.2: Boolean operators

```
Z> find @or information retrieval
```

Querying for the intersection of all documents containing the terms *information* AND *retrieval*: The hit set is a subset of the corresponding OR query.

```
Z> find @and information retrieval
```

Querying for the intersection of all documents containing the terms *information* AND *retrieval*, taking proximity into account: The hit set is a subset of the corresponding AND query (see the PQF grammar for details on the proximity operator):

```
Z> find @prox 0 3 0 2 k 2 information retrieval
```

Querying for the intersection of all documents containing the terms *information* AND *retrieval*, in the same order and near each other as described in the term list. The hit set is a subset of the corresponding PROXIMITY query.

```
Z> find "information retrieval"
```

5.2.1.3 Atomic queries (APT)

Atomic queries are the query parts which work on one access point only. These consist of *an attribute list* followed by a *single term* or a *quoted term list*, and are often called *Attributes-Plus-Terms (APT)* queries.

Atomic (APT) queries are always leaf nodes in the PQF query tree. UN-supplied non-use attributes types 2-12 are either inherited from higher nodes in the query tree, or are set to Zebra's default values. See Section 5.2.3 for details.

Querying for the term *information* in the default index using the default attribute set, the server choice of access point/index, and the default non-use attributes.

Name	Type	Notes
attribute list	List of *orthogonal* attributes	Any of the orthogonal attribute types may be omitted, these are inherited from higher query tree nodes, or if not inherited, are set to the default Zebra configuration values.
term	single *term* or *quoted term list*	Here the search terms or list of search terms is added to the query

Table 5.3: Atomic queries (APT)

```
Z> find information
```

Equivalent query fully specified including all default values:

```
Z> find @attrset bib-1 @attr 1=1017 @attr 2=3 @attr 3=3 @attr 4=1  ↩
   @attr 5=100 @attr 6=1 information
```

Finding all documents which have the term *debussy* in the title field.

```
Z> find @attr 1=4 debussy
```

The *scan* operation is only supported with atomic APT queries, as it is bound to one access point at a time. Boolean query trees are not allowed during *scan*.

For example, we might want to scan the title index, starting with the term *debussy*, and displaying this and the following terms in lexicographic order:

```
Z> scan @attr 1=4 debussy
```

5.2.1.4 Named Result Sets

Named result sets are supported in Zebra, and result sets can be used as operands without limitations. It follows that named result sets are leaf nodes in the PQF query tree, exactly as atomic APT queries are.

After the execution of a search, the result set is available at the server, such that the client can use it for subsequent searches or retrieval requests. The Z30.50 standard actually stresses the fact that result sets are volatile. It may cease to exist at any time point after search, and the server will send a diagnostic to the effect that the requested result set does not exist any more.

Defining a named result set and re-using it in the next query, using yaz-client. Notice that the client, not the server, assigns the string '1' to the named result set.

```
Z> f @attr 1=4 mozart
...
Number of hits: 43, setno 1
...
```

```
Z> f @and @set 1 @attr 1=4 amadeus
...
Number of hits: 14, setno 2
```

Note
Named result sets are only supported by the Z39.50 protocol. The SRU web service is stateless, and therefore the notion of named result sets does not exist when accessing a Zebra server by the SRU protocol.

5.2.1.5 Zebra's special access point of type 'string'

The numeric *use (type 1)* attribute is usually referred to from a given attribute set. In addition, Zebra let you use *any internal index name defined in your configuration* as use attribute value. This is a great feature for debugging, and when you do not need the complexity of defined use attribute values. It is the preferred way of accessing Zebra indexes directly.

Finding all documents which have the term list "information retrieval" in an Zebra index, using its internal full string name. Scanning the same index.

```
Z> find @attr 1=sometext "information retrieval"
Z> scan @attr 1=sometext aterm
```

Searching or scanning the bib-1 use attribute 54 using its string name:

```
Z> find @attr 1=Code-language eng
Z> scan @attr 1=Code-language ""
```

It is possible to search in any silly string index - if it's defined in your indexing rules and can be parsed by the PQF parser. This is definitely not the recommended use of this facility, as it might confuse your users with some very unexpected results.

```
Z> find @attr 1=silly/xpath/alike[@index]/name "information  ↩
   retrieval"
```

See also Section 5.3.5 for details, and the section called "The SRU Server" for the SRU PQF query extension using string names as a fast debugging facility.

5.2.1.6 Zebra's special access point of type 'XPath' for GRS-1 filters

As we have seen above, it is possible (albeit seldom a great idea) to emulate XPath 1.0 based search by defining *use (type 1) string* attributes which in appearance *resemble XPath queries*. There are two problems with this approach: first, the XPath-look-alike has to be defined at indexing time, no new undefined XPath queries can entered at search time, and second, it might confuse users very much that an XPath-alike index name in fact gets populated from a possible entirely different XML element than it pretends to access.

When using the GRS-1 Record Model (see Chapter 9), we have the possibility to embed *life* XPath expressions in the PQF queries, which are here called *use (type 1) xpath* attributes. You must enable the `xpath enable` directive in your `.abs` configuration files.

Note
Only a *very* restricted subset of the XPath 1.0 standard is supported as the GRS-1 record model is simpler than a full XML DOM structure. See the following examples for possibilities.

Finding all documents which have the term "content" inside a text node found in a specific XML DOM *subtree*, whose starting element is addressed by XPath.

```
Z> find @attr 1=/root content
Z> find @attr 1=/root/first content
```

Notice that the XPath must be absolute, i.e., must start with '/', and that the XPath `descendant-or-self` *axis followed by a text node selection* `text()` *is implicitly appended to the stated XPath.* It follows that the above searches are interpreted as:

```
Z> find @attr 1=/root//text() content
Z> find @attr 1=/root/first//text() content
```

Searching inside attribute strings is possible:

```
Z> find @attr 1=/link/@creator morten
```

Filter the addressing XPath by a predicate working on exact string values in attributes (in the XML sense) can be done: return all those docs which have the term "english" contained in one of all text sub nodes of the subtree defined by the XPath `/record/title[@lang='en']`. And similar predicate filtering.

```
Z> find @attr 1=/record/title[@lang='en'] english
Z> find @attr 1=/link[@creator='sisse'] sibelius
Z> find @attr 1=/link[@creator='sisse']/description[@xml:lang='da']  ↩
   sibelius
```

Combining numeric indexes, boolean expressions, and xpath based searches is possible:

```
Z> find @attr 1=/record/title @and foo bar
Z> find @and @attr 1=/record/title foo @attr 1=4 bar
```

Escaping PQF keywords and other non-parseable XPath constructs with `'{ }'` to prevent client-side PQF parsing syntax errors:

```
Z> find @attr {1=/root/first[@attr='danish']} content
Z> find @attr {1=/record/@set} oai
```

Warning
It is worth mentioning that these dynamic performed XPath queries are a performance bottleneck, as no optimized specialized indexes can be used. Therefore, avoid the use of this facility when speed is essential, and the database content size is medium to large.

5.2.2 Explain Attribute Set

The Z39.50 standard defines the Explain attribute set Exp-1, which is used to discover information about a server's search semantics and functional capabilities Zebra exposes a "classic" Explain database by base name `IR-Explain-1`, which is populated with system internal information.

The attribute-set `exp-1` consists of a single use attribute (type 1).

In addition, the non-Use BIB-1 attributes, that is, the types *Relation*, *Position*, *Structure*, *Truncation*, and *Completeness* are imported from the BIB-1 attribute set, and may be used within any explain query.

5.2.2.1 Use Attributes (type = 1)

The following Explain search attributes are supported: `ExplainCategory` (@attr 1=1), `DatabaseName` (@attr 1=3), `DateAdded` (@attr 1=9), `DateChanged`(@attr 1=10).

A search in the use attribute `ExplainCategory` supports only these predefined values: `CategoryList`, `TargetInfo`, `DatabaseInfo`, `AttributeDetails`.

See `tab/explain.att` and the Z39.50 standard for more information.

5.2.2.2 Explain searches with yaz-client

Classic Explain only defines retrieval of Explain information via ASN.1. Practically no Z39.50 clients supports this. Fortunately they don't have to - Zebra allows retrieval of this information in other formats: `SUTRS`, `XML`, `GRS-1` and `ASN.1` Explain.

List supported categories to find out which explain commands are supported:

```
Z> base IR-Explain-1
Z> find @attr exp1 1=1 categorylist
Z> form sutrs
Z> show 1+2
```

Get target info, that is, investigate which databases exist at this server endpoint:

```
Z> base IR-Explain-1
Z> find @attr exp1 1=1 targetinfo
Z> form xml
Z> show 1+1
Z> form grs-1
Z> show 1+1
Z> form sutrs
Z> show 1+1
```

List all supported databases, the number of hits is the number of databases found, which most commonly are the following two: the `Default` and the `IR-Explain-1` databases.

```
Z> base IR-Explain-1
Z> find @attr exp1 1=1 databaseinfo
Z> form sutrs
Z> show 1+2
```

Get database info record for database `Default`.

```
Z> base IR-Explain-1
Z> find @and @attr exp1 1=1 databaseinfo @attr exp1 1=3 Default
```

Identical query with explicitly specified attribute set:

```
Z> base IR-Explain-1
Z> find @attrset exp1 @and @attr 1=1 databaseinfo @attr 1=3 Default
```

Get attribute details record for database `Default`. This query is very useful to study the internal Zebra indexes. If records have been indexed using the `alvis` XSLT filter, the string representation names of the known indexes can be found.

```
Z> base IR-Explain-1
Z> find @and @attr exp1 1=1 attributedetails @attr exp1 1=3 Default
```

Identical query with explicitly specified attribute set:

```
Z> base IR-Explain-1
Z> find @attrset exp1 @and @attr 1=1 attributedetails @attr 1=3  ↩
   Default
```

5.2.3 BIB-1 Attribute Set

Most of the information contained in this section is an excerpt of the ATTRIBUTE SET BIB-1 (Z39.50-1995) SEMANTICS found at . The BIB-1 Attribute Set Semantics from 1995, also in an updated BIB-1 Attribute Set version from 2003. Index Data is not the copyright holder of this information, except for the configuration details, the listing of Zebra's capabilities, and the example queries.

5.2.3.1 Use Attributes (type 1)

A use attribute specifies an access point for any atomic query. These access points are highly dependent on the attribute set used in the query, and are user configurable using the following default configuration files: `tab/bib1.att`, `tab/dan1.att`, `tab/explain.att`, and `tab/gils.att`.

For example, some few BIB-1 use attributes from the `tab/bib1.att` are:

```
att 1               Personal-name
att 2               Corporate-name
att 3               Conference-name
att 4               Title
...
att 1009            Subject-name-personal
att 1010            Body-of-text
att 1011            Date/time-added-to-db
...
att 1016            Any
att 1017            Server-choice
att 1018            Publisher
```

```
...
att 1035            Anywhere
att 1036            Author-Title-Subject
```

New attribute sets can be added by adding new `tab/*.att` configuration files, which need to be sourced in the main configuration `zebra.cfg`.

In addition, Zebra allows the access of *internal index names* and *dynamic XPath* as use attributes; see Section 5.2.1.5 and Section 5.2.1.6.

Phrase search for *information retrieval* in the title-register, scanning the same register afterwards:

```
Z> find @attr 1=4 "information retrieval"
Z> scan @attr 1=4 information
```

5.2.4 Zebra general Bib1 Non-Use Attributes (type 2-6)

5.2.4.1 Relation Attributes (type 2)

Relation attributes describe the relationship of the access point (left side of the relation) to the search term as qualified by the attributes (right side of the relation), e.g., Date-publication <= 1975.

Relation	Value	Notes
Less than	1	supported
Less than or equal	2	supported
Equal	3	default
Greater or equal	4	supported
Greater than	5	supported
Not equal	6	unsupported
Phonetic	100	unsupported
Stem	101	unsupported
Relevance	102	supported
AlwaysMatches	103	supported *

Table 5.4: Relation Attributes (type 2)

Note
AlwaysMatches searches are only supported if alwaysmatches indexing has been enabled. See Section 10.1

The relation attributes 1-5 are supported and work exactly as expected. All ordering operations are based on a lexicographical ordering, *except* when the structure attribute numeric (109) is used. In this case, ordering is numerical. See Section 5.2.4.3.

```
Z> find @attr 1=Title @attr 2=1 music
...
Number of hits: 11745, setno 1
...
Z> find @attr 1=Title @attr 2=2 music
...
Number of hits: 11771, setno 2
...
Z> find @attr 1=Title @attr 2=3 music
...
Number of hits: 532, setno 3
...
Z> find @attr 1=Title @attr 2=4 music
...
Number of hits: 11463, setno 4
...
Z> find @attr 1=Title @attr 2=5 music
...
Number of hits: 11419, setno 5
```

The relation attribute *Relevance (102)* is supported, see Section 6.9 for full information.

Ranked search for *information retrieval* in the title-register:

```
Z> find @attr 1=4 @attr 2=102 "information retrieval"
```

The relation attribute *AlwaysMatches (103)* is in the default configuration supported in conjecture with structure attribute *Phrase (1)* (which may be omitted by default). It can be configured to work with other structure attributes, see the configuration file `tab/default.idx` and Section 5.3.5.

AlwaysMatches (103) is a great way to discover how many documents have been indexed in a given field. The search term is ignored, but needed for correct PQF syntax. An empty search term may be supplied.

```
Z> find @attr 1=Title  @attr 2=103  ""
Z> find @attr 1=Title  @attr 2=103  @attr 4=1 ""
```

5.2.4.2 Position Attributes (type 3)

The position attribute specifies the location of the search term within the field or subfield in which it appears.

Position	Value	Notes
First in field	1	supported *
First in subfield	2	supported *
Any position in field	3	default

Table 5.5: Position Attributes (type 3)

Note
Zebra only supports first-in-field seaches if the `firstinfield` is enabled for the index Refer to Section 10.1. Zebra does not distinguish between first in field and first in subfield. They result in the same hit count. Searching for first position in (sub)field in only supported in Zebra 2.0.2 and later.

5.2.4.3 Structure Attributes (type 4)

The structure attribute specifies the type of search term. This causes the search to be mapped on different Zebra internal indexes, which must have been defined at index time.

The possible values of the `structure attribute (type 4)` can be defined using the configuration file `tab/default.idx`. The default configuration is summarized in this table.

Structure	Value	Notes
Phrase	1	default
Word	2	supported
Key	3	supported
Year	4	supported
Date (normalized)	5	supported
Word list	6	supported
Date (un-normalized)	100	unsupported
Name (normalized)	101	unsupported
Name (un-normalized)	102	unsupported
Structure	103	unsupported
Urx	104	supported
Free-form-text	105	supported
Document-text	106	supported
Local-number	107	supported
String	108	unsupported
Numeric string	109	supported

Table 5.6: Structure Attributes (type 4)

The structure attribute values `Word list (6)` is supported, and maps to the boolean `AND` combination of words supplied. The word list is useful when Google-like bag-of-word queries need to be translated from a GUI query language to PQF. For example, the following queries are equivalent:

```
Z> find @attr 1=Title @attr 4=6 "mozart amadeus"
Z> find @attr 1=Title  @and mozart amadeus
```

The structure attribute value `Free-form-text (105)` and `Document-text (106)` are supported, and map both to the boolean `OR` combination of words supplied. The following queries are equivalent:

```
Z> find @attr 1=Body-of-text @attr 4=105 "bach salieri teleman"
Z> find @attr 1=Body-of-text @attr 4=106 "bach salieri teleman"
Z> find @attr 1=Body-of-text @or bach @or salieri teleman
```

This `OR` list of terms is very useful in combination with relevance ranking:

```
Z> find @attr 1=Body-of-text @attr 2=102 @attr 4=105 "bach salieri  ↩
   teleman"
```

The structure attribute value `Local number (107)` is supported, and maps always to the Zebra internal document ID, irrespectively which use attribute is specified. The following queries have exactly the same unique record in the hit set:

```
Z> find @attr 4=107 10
Z> find @attr 1=4 @attr 4=107 10
Z> find @attr 1=1010 @attr 4=107 10
```

In the GILS schema (`gils.abs`), the west-bounding-coordinate is indexed as type `n`, and is therefore searched by specifying *structure=Numeric String*. To match all those records with west-bounding-coordinate greater than -114 we use the following query:

```
Z> find @attr 4=109 @attr 2=5 @attr gils 1=2038 -114
```

Note
The exact mapping between PQF queries and Zebra internal indexes and index types is explained in Section 5.3.5.

5.2.4.4 Truncation Attributes (type = 5)

The truncation attribute specifies whether variations of one or more characters are allowed between search term and hit terms, or not. Using non-default truncation attributes will broaden the document hit set of a search query.

Truncation	Value	Notes
Right truncation	1	supported
Left truncation	2	supported
Left and right truncation	3	supported
Do not truncate	100	default
Process # in search term	101	supported
RegExpr-1	102	supported
RegExpr-2	103	supported

Table 5.7: Truncation Attributes (type 5)

The truncation attribute values 1-3 perform the obvious way:

```
Z> scan @attr 1=Body-of-text  schnittke
...
* schnittke (81)
schnittkes (31)
schnittstelle (1)
...
```

```
Z> find @attr 1=Body-of-text  @attr 5=1 schnittke
...
Number of hits: 95, setno 7
...
Z> find @attr 1=Body-of-text  @attr 5=2 schnittke
...
Number of hits: 81, setno 6
...
Z> find @attr 1=Body-of-text  @attr 5=3 schnittke
...
Number of hits: 95, setno 8
```

The truncation attribute value `Process # in search term (101)` is a poor-man's regular expression search. It maps each `#` to `.*`, and performs then a `Regexp-1 (102)` regular expression search. The following two queries are equivalent:

```
Z> find @attr 1=Body-of-text  @attr 5=101 schnit#ke
Z> find @attr 1=Body-of-text  @attr 5=102 schnit.*ke
...
Number of hits: 89, setno 10
```

The truncation attribute value `Regexp-1 (102)` is a normal regular search, see Section 5.3.6 for details.

```
Z> find @attr 1=Body-of-text  @attr 5=102 schnit+ke
Z> find @attr 1=Body-of-text  @attr 5=102 schni[a-t]+ke
```

The truncation attribute value `Regexp-2 (103)` is a Zebra specific extension which allows *fuzzy* matches. One single error in spelling of search terms is allowed, i.e., a document is hit if it includes a term which can be mapped to the used search term by one character substitution, addition, deletion or change of position.

```
Z> find @attr 1=Body-of-text  @attr 5=100 schnittke
...
Number of hits: 81, setno 14
...
Z> find @attr 1=Body-of-text  @attr 5=103 schnittke
...
Number of hits: 103, setno 15
...
```

5.2.4.5 Completeness Attributes (type = 6)

The `Completeness Attributes (type = 6)` is used to specify that a given search term or term list is either part of the terms of a given index/field (`Incomplete subfield (1)`), or is what literally is found in the entire field's index (`Complete field (3)`).

The `Completeness Attributes (type = 6)` is only partially and conditionally supported in the sense that it is ignored if the hit index is not of structure `type="w"` or `type="p"`.

`Incomplete subfield (1)` is the default, and makes Zebra use register `type="w"`, whereas `Complete field (3)` triggers search and scan in index `type="p"`.

Completeness	Value	Notes
Incomplete subfield	1	default
Complete subfield	2	deprecated
Complete field	3	supported

Table 5.8: Completeness Attributes (type = 6)

The `Complete subfield (2)` is a reminiscent from the happy `MARC` binary format days. Zebra does not support it, but maps silently to `Complete field (3)`.

Note
The exact mapping between PQF queries and Zebra internal indexes and index types is explained in Section 5.3.5.

5.3 Extended Zebra RPN Features

The Zebra internal query engine has been extended to specific needs not covered by the `bib-1` attribute set query model. These extensions are *non-standard* and *non-portable*: most functional extensions are modeled over the `bib-1` attribute set, defining type 7 and higher values. There are also the special `string` type index names for the `idxpath` attribute set.

5.3.1 Zebra specific retrieval of all records

Zebra defines a hardwired `string` index name called `_ALLRECORDS`. It matches any record contained in the database, if used in conjunction with the relation attribute `AlwaysMatches (103)`.

The `_ALLRECORDS` index name is used for total database export. The search term is ignored, it may be empty.

```
Z> find @attr 1=_ALLRECORDS @attr 2=103 ""
```

Combination with other index types can be made. For example, to find all records which are *not* indexed in the `Title` register, issue one of the two equivalent queries:

```
Z> find @not @attr 1=_ALLRECORDS @attr 2=103 "" @attr 1=Title @attr ↩
   2=103 ""
Z> find @not @attr 1=_ALLRECORDS @attr 2=103 "" @attr 1=4 @attr 2=103 ↩
    ""
```

Warning
The special string index `_ALLRECORDS` is experimental, and the provided functionality and syntax may very well change in future releases of Zebra.

5.3.2 Zebra specific Search Extensions to all Attribute Sets

Zebra extends the BIB-1 attribute types, and these extensions are recognized regardless of attribute set used in a `search` operation query.

Name	Value	Operation	Zebra version
Embedded Sort	7	search	1.1
Term Set	8	search	1.1
Rank Weight	9	search	1.1
Term Reference	10	search	1.4
Local Approx Limit	11	search	1.4
Global Approx Limit	12	search	2.0.8
Maximum number of truncated terms (truncmax)	13	search	2.0.10
Specifies whether un-indexed fields should be ignored. A zero value (default) throws a diagnostic when an un-indexed field is specified. A non-zero value makes it return 0 hits.	14	search	2.0.16

Table 5.9: Zebra Search Attribute Extensions

5.3.2.1 Zebra Extension Embedded Sort Attribute (type 7)

The embedded sort is a way to specify sort within a query - thus removing the need to send a Sort Request separately. It is both faster and does not require clients to deal with the Sort Facility.

All ordering operations are based on a lexicographical ordering, *except* when the `structure attribute numeric (109)` is used. In this case, ordering is numerical. See Section 5.2.4.3.

The possible values after attribute `type 7` are `1` ascending and `2` descending. The attributes+term (APT) node is separate from the rest and must be `@or`'ed. The term associated with APT is the sorting level in integers, where `0` means primary sort, `1` means secondary sort, and so forth. See also Section 6.9.

For example, searching for water, sort by title (ascending)

```
Z> find @or @attr 1=1016 water @attr 7=1 @attr 1=4 0
```

Or, searching for water, sort by title ascending, then date descending

```
Z> find @or @or @attr 1=1016 water @attr 7=1 @attr 1=4 0 @attr 7=2  ↩
   @attr 1=30 1
```

5.3.2.2 Zebra Extension Rank Weight Attribute (type 9)

Rank weight is a way to pass a value to a ranking algorithm - so that one APT has one value - while another as a different one. See also Section 6.9.

For example, searching for utah in title with weight 30 as well as any with weight 20:

```
Z> find @attr 2=102 @or @attr 9=30 @attr 1=4 utah @attr 9=20 utah
```

5.3.2.3 Zebra Extension Term Reference Attribute (type 10)

Zebra supports the searchResult-1 facility. If the Term Reference Attribute (type 10) is given, that specifies a subqueryId value returned as part of the search result. It is a way for a client to name an APT part of a query.

Warning
Experimental. Do not use in production code.

5.3.2.4 Local Approximative Limit Attribute (type 11)

Zebra computes - unless otherwise configured - the exact hit count for every APT (leaf) in the query tree. These hit counts are returned as part of the searchResult-1 facility in the binary encoded Z39.50 search response packages.

By setting an estimation limit size of the resultset of the APT leaves, Zebra stops processing the result set when the limit length is reached. Hit counts under this limit are still precise, but hit counts over it are estimated using the statistics gathered from the chopped result set.

Specifying a limit of `0` results in exact hit counts.

For example, we might be interested in exact hit count for a, but for b we allow hit count estimates for 1000 and higher.

```
Z> find @and a @attr 11=1000 b
```

Note
The estimated hit count facility makes searches faster, as one only needs to process large hit lists partially. It is mostly used in huge databases, where you you want trade exactness of hit counts against speed of execution.

Warning
Do not use approximative hit count limits in conjunction with relevance ranking, as re-sorting of the result set only works when the entire result set has been processed.

5.3.2.5 Global Approximative Limit Attribute (type 12)

By default Zebra computes precise hit counts for a query as a whole. Setting attribute 12 makes it perform approximative hit counts instead. It has the same semantics as `estimatehits` for the Section 6.2.

The attribute (12) can occur anywhere in the query tree. Unlike regular attributes it does not relate to the leaf (APT) - but to the whole query.

Warning
Do not use approximative hit count limits in conjunction with relevance ranking, as re-sorting of the result set only works when the entire result set has been processed.

5.3.3 Zebra specific Scan Extensions to all Attribute Sets

Zebra extends the Bib1 attribute types, and these extensions are recognized regardless of attribute set used in a scan operation query.

Name	Type	Operation	Zebra version
Result Set Narrow	8	scan	1.3
Approximative Limit	12	scan	2.0.20

Table 5.10: Zebra Scan Attribute Extensions

5.3.3.1 Zebra Extension Result Set Narrow (type 8)

If attribute Result Set Narrow (type 8) is given for scan, the value is the name of a result set. Each hit count in scan is `@and`'ed with the result set given.

Consider for example the case of scanning all title fields around the scanterm *mozart*, then refining the scan by issuing a filtering query for *amadeus* to restrict the scan to the result set of the query:

```
Z> scan @attr 1=4 mozart
...
* mozart (43)
  mozartforskningen (1)
  mozartiana (1)
  mozarts (16)
...
Z> f @attr 1=4 amadeus
...
Number of hits: 15, setno 2
...
Z> scan @attr 1=4 @attr 8=2 mozart
...
* mozart (14)
  mozartforskningen (0)
```

```
        mozartiana (0)
        mozarts (1)
      ...
```

Zebra 2.0.2 and later is able to skip 0 hit counts. This, however, is known not to scale if the number of terms to skip is high. This most likely will happen if the result set is small (and result in many 0 hits).

5.3.3.2 Zebra Extension Approximative Limit (type 12)

The Zebra Extension Approximative Limit (type 12) is a way to enable approximate hit counts for scan hit counts, in the same way as for search hit counts.

5.3.4 Zebra special IDXPATH Attribute Set for GRS-1 indexing

The attribute-set `idxpath` consists of a single Use (type 1) attribute. All non-use attributes behave as normal.

This feature is enabled when defining the `xpath enable` option in the GRS-1 filter `*.abs` configuration files. If one wants to use the special `idxpath` numeric attribute set, the main Zebra configuration file `zebra.cfg` directive `attset: idxpath.att` must be enabled.

Warning
The `idxpath` is deprecated, may not be supported in future Zebra versions, and should definitely not be used in production code.

5.3.4.1 IDXPATH Use Attributes (type = 1)

This attribute set allows one to search GRS-1 filter indexed records by XPATH like structured index names.

Warning
The `idxpath` option defines hard-coded index names, which might clash with your own index names.

See `tab/idxpath.att` for more information.

Search for all documents starting with root element `/root` (either using the numeric or the string use attributes):

```
Z> find @attrset idxpath @attr 1=1 @attr 4=3 root/
Z> find @attr idxpath 1=1 @attr 4=3 root/
Z> find @attr 1=_XPATH_BEGIN @attr 4=3 root/
```

Search for all documents where specific nested XPATH `/c1/c2/../cn` exists. Notice the very counter-intuitive *reverse* notation!

IDXPATH	Value	String Index	Notes
XPATH Begin	1	_XPATH_BEGIN	deprecated
XPATH End	2	_XPATH_END	deprecated
XPATH CData	1016	_XPATH_CDATA	deprecated
XPATH Attribute Name	3	_XPATH_ATTR_NAME	deprecated
XPATH Attribute CData	1015	_XPATH_ATTR_CDATA	deprecated

Table 5.11: Zebra specific IDXPATH Use Attributes (type 1)

```
Z> find @attrset idxpath @attr 1=1 @attr 4=3 cn/cn-1/../c1/
Z> find @attr 1=_XPATH_BEGIN @attr 4=3 cn/cn-1/../c1/
```

Search for CDATA string *text* in any element

```
Z> find @attrset idxpath @attr 1=1016 text
Z> find @attr 1=_XPATH_CDATA text
```

Search for CDATA string *anothertext* in any attribute:

```
Z> find @attrset idxpath @attr 1=1015 anothertext
Z> find @attr 1=_XPATH_ATTR_CDATA anothertext
```

Search for all documents with have an XML element node including an XML attribute named *creator*

```
Z> find @attrset idxpath @attr 1=3 @attr 4=3 creator
Z> find @attr 1=_XPATH_ATTR_NAME @attr 4=3 creator
```

Combining usual `bib-1` attribute set searches with `idxpath` attribute set searches:

```
Z> find @and @attr idxpath 1=1 @attr 4=3 link/ @attr 1=4 mozart
Z> find @and @attr 1=_XPATH_BEGIN @attr 4=3 link/ @attr 1= ↩
   _XPATH_CDATA mozart
```

Scanning is supported on all `idxpath` indexes, both specified as numeric use attributes, or as string index names.

```
Z> scan  @attrset idxpath @attr 1=1016 text
Z> scan  @attr 1=_XPATH_ATTR_CDATA anothertext
Z> scan  @attrset idxpath @attr 1=3 @attr 4=3 ''
```

5.3.5 Mapping from PQF atomic APT queries to Zebra internal register indexes

The rules for PQF APT mapping are rather tricky to grasp in the first place. We deal first with the rules for deciding which internal register or string index to use, according to the use attribute or access point specified in the query. Thereafter we deal with the rules for determining the correct structure type of the named register.

5.3.5.1 Mapping of PQF APT access points

Zebra understands four fundamental different types of access points, of which only the *numeric use attribute* type access points are defined by the Z39.50 standard. All other access point types are Zebra specific, and non-portable.

Access Point	Type	Grammar	Notes
Use attribute	numeric	[1-9][1-9]*	directly mapped to string index name
String index name	string	[a-zA-Z](\-?[a-zA-Z0-9])*	normalized name is used as internal string index name
Zebra internal index name	zebra	_[a-zA-Z](_?[a-zA-Z0-9])*	hardwired internal string index name
XPATH special index	XPath	/.*	special xpath search for GRS-1 indexed records

Table 5.12: Access point name mapping

`Attribute set names` and `string index names` are normalizes according to the following rules: all *single* hyphens `'-'` are stripped, and all upper case letters are folded to lower case.

Numeric use attributes are mapped to the Zebra internal string index according to the attribute set definition in use. The default attribute set is `BIB-1`, and may be omitted in the PQF query.

According to normalization and numeric use attribute mapping, it follows that the following PQF queries are considered equivalent (assuming the default configuration has not been altered):

```
Z> find  @attr 1=Body-of-text serenade
Z> find  @attr 1=bodyoftext serenade
Z> find  @attr 1=BodyOfText serenade
Z> find  @attr 1=bO-d-Y-of-tE-x-t serenade
Z> find  @attr 1=1010 serenade
Z> find  @attrset BIB-1 @attr 1=1010 serenade
Z> find  @attrset bib1 @attr 1=1010 serenade
Z> find  @attrset Bib1 @attr 1=1010 serenade
Z> find  @attrset b-I-b-1 @attr 1=1010 serenade
```

The *numerical* `use attributes (type 1)` are interpreted according to the attribute sets which have been loaded in the `zebra.cfg` file, and are matched against specific fields as specified in the `.abs` file which describes the profile of the records which have been loaded. If no use attribute is provided, a default of BIB-1 Use Any (1016) is assumed. The predefined use attribute sets can be reconfigured by tweaking the configuration files `tab/*.att`, and new attribute sets can be defined by adding similar files in the configuration path `profilePath` of the server.

String indexes can be accessed directly, independently which attribute set is in use. These are just ignored. The above mentioned name normalization applies. String index names are defined in the used indexing filter configuration files, for example in the `GRS-1 *.abs` configuration files, or in the `alvis` filter XSLT indexing stylesheets.

Zebra internal indexes can be accessed directly, according to the same rules as the user defined string indexes. The only difference is that Zebra internal index names are hardwired, all uppercase and must start with the character `'_'`.

Finally, `XPATH` access points are only available using the `GRS-1` filter for indexing. These access point names must start with the character `'/'`, they are *not normalized*, but passed unaltered to the Zebra internal XPATH engine. See Section 5.2.1.6.

5.3.5.2 Mapping of PQF APT structure and completeness to register type

Internally Zebra has in its default configuration several different types of registers or indexes, whose tokenization and character normalization rules differ. This reflects the fact that searching fundamental different tokens like dates, numbers, bitfields and string based text needs different rule sets.

If a *Structure* attribute of *Phrase* is used in conjunction with a *Completeness* attribute of *Complete (Sub)field*, the term is matched against the contents of the phrase (long word) register, if one exists for the given *Use* attribute. A phrase register is created for those fields in the GRS-1 `*.abs` file that contains a `p`-specifier.

```
Z> scan @attr 1=Title @attr 4=1 @attr 6=3 beethoven
...
bayreuther festspiele (1)
* beethoven bibliography database (1)
benny carter (1)
...
Z> find @attr 1=Title @attr 4=1 @attr 6=3 "beethoven bibliography"
...
Number of hits: 0, setno 5
...
Z> find @attr 1=Title @attr 4=1 @attr 6=3 "beethoven bibliography  ↩
   database"
...
Number of hits: 1, setno 6
```

If *Structure=Phrase* is used in conjunction with *Incomplete Field* - the default value for *Completeness*, the search is directed against the normal word registers, but if the term contains multiple words, the term will only match if all of the words are found immediately adjacent, and in the given order. The word search is performed on those fields that are indexed as type `w` in the GRS-1 `*.abs` file.

```
Z> scan @attr 1=Title @attr 4=1 @attr 6=1 beethoven
...
  beefheart (1)
* beethoven (18)
  beethovens (7)
...
Z> find @attr 1=Title @attr 4=1 @attr 6=1 beethoven
...
Number of hits: 18, setno 1
...
Z> find @attr 1=Title @attr 4=1 @attr 6=1 "beethoven  bibliography"
...
Number of hits: 2, setno 2
```

Structure	Completeness	Register type	Notes
phrase (@attr 4=1), word (@attr 4=2), word-list (@attr 4=6), free-form-text (@attr 4=105), or document-text (@attr 4=106)	Incomplete field (@attr 6=1)	Word ('w')	Traditional tokenized and character normalized word index
phrase (@attr 4=1), word (@attr 4=2), word-list (@attr 4=6), free-form-text (@attr 4=105), or document-text (@attr 4=106)	complete field' (@attr 6=3)	Phrase ('p')	Character normalized, but not tokenized index for phrase matches
urx (@attr 4=104)	ignored	URX/URL ('u')	Special index for URL web addresses
numeric (@attr 4=109)	ignored	Numeric ('n')	Special index for digital numbers
key (@attr 4=3)	ignored	Null bitmap ('0')	Used for non-tokenized and non-normalized bit sequences
year (@attr 4=4)	ignored	Year ('y')	Non-tokenized and non-normalized 4 digit numbers
date (@attr 4=5)	ignored	Date ('d')	Non-tokenized and non-normalized ISO date strings
ignored	ignored	Sort ('s')	Used with special sort attribute set (@attr 7=1, @attr 7=2)
overruled	overruled	special	Internal record ID register, used whenever Relation Always Matches (@attr 2=103) is specified

Table 5.13: Structure and completeness mapping to register types

```
...
```

If the *Structure* attribute is *Word List*, *Free-form Text*, or *Document Text*, the term is treated as a natural-language, relevance-ranked query. This search type uses the word register, i.e. those fields that are indexed as type `w` in the GRS-1 `*.abs` file.

If the *Structure* attribute is *Numeric String* the term is treated as an integer. The search is performed on those fields that are indexed as type `n` in the GRS-1 `*.abs` file.

If the *Structure* attribute is *URX* the term is treated as a URX (URL) entity. The search is performed on those fields that are indexed as type `u` in the `*.abs` file.

If the *Structure* attribute is *Local Number* the term is treated as native Zebra Record Identifier.

If the *Relation* attribute is *Equals* (default), the term is matched in a normal fashion (modulo truncation and processing of individual words, if required). If *Relation* is *Less Than*, *Less Than or Equal*, *Greater than*, or *Greater than or Equal*, the term is assumed to be numerical, and a standard regular expression is constructed to match the given expression. If *Relation* is *Relevance*, the standard natural-language query processor is invoked.

For the *Truncation* attribute, *No Truncation* is the default. *Left Truncation* is not supported. *Process # in search term* is supported, as is *Regxp-1*. *Regxp-2* enables the fault-tolerant (fuzzy) search. As a default, a single error (deletion, insertion, replacement) is accepted when terms are matched against the register contents.

5.3.6 Zebra Regular Expressions in Truncation Attribute (type = 5)

Each term in a query is interpreted as a regular expression if the truncation value is either *Regxp-1 (@attr 5=102)* or *Regxp-2 (@attr 5=103)*. Both query types follow the same syntax with the operands:

`x`	Matches the character `x`.
`.`	Matches any character.
`[ .. ]`	Matches the set of characters specified; such as `[abc]` or `[a-c]`.

Table 5.14: Regular Expression Operands

The above operands can be combined with the following operators:

`x*`	Matches `x` zero or more times. Priority: high.
`x+`	Matches `x` one or more times. Priority: high.
`x?`	Matches `x` zero or once. Priority: high.
`xy`	Matches `x`, then `y`. Priority: medium.
`x|y`	Matches either `x` or `y`. Priority: low.
`( )`	The order of evaluation may be changed by using parentheses.

Table 5.15: Regular Expression Operators

If the first character of the `Regxp-2` query is a plus character (+) it marks the beginning of a section with non-standard specifiers. The next plus character marks the end of the section. Currently Zebra only supports one specifier, the error tolerance, which consists one digit.

Since the plus operator is normally a suffix operator the addition to the query syntax doesn't violate the syntax for standard regular expressions.

For example, a phrase search with regular expressions in the title-register is performed like this:

```
Z> find @attr 1=4 @attr 5=102 "informat.* retrieval"
```

Combinations with other attributes are possible. For example, a ranked search with a regular expression:

```
Z> find @attr 1=4 @attr 5=102 @attr 2=102 "informat.* retrieval"
```

5.4 Server Side CQL to PQF Query Translation

Using the `<cql2rpn>l2rpn.txt</cql2rpn>` YAZ Frontend Virtual Hosts option, one can configure the YAZ Frontend CQL-to-PQF converter, specifying the interpretation of various CQL indexes, relations, etc. in terms of Type-1 query attributes.

For example, using server-side CQL-to-PQF conversion, one might query a zebra server like this:

```
yaz-client localhost:9999
Z> querytype cql
Z> find text=(plant and soil)
```

and - if properly configured - even static relevance ranking can be performed using CQL query syntax:

```
Z> find text = /relevant (plant and soil)
```

By the way, the same configuration can be used to search using client-side CQL-to-PQF conversion: (the only difference is `querytype cql2rpn` instead of `querytype cql`, and the call specifying a local conversion file)

```
yaz-client -q local/cql2pqf.txt localhost:9999
Z> querytype cql2rpn
Z> find text=(plant and soil)
```

Exhaustive information can be found in the Section CQL to RPN conversion" in the YAZ manual.

Chapter 6

Administrating Zebra

Unlike many simpler retrieval systems, Zebra supports safe, incremental updates to an existing index.

Normally, when Zebra modifies the index it reads a number of records that you specify. Depending on your specifications and on the contents of each record one the following events take place for each record:

Insert The record is indexed as if it never occurred before. Either the Zebra system doesn't know how to identify the record or Zebra can identify the record but didn't find it to be already indexed.

Modify The record has already been indexed. In this case either the contents of the record or the location (file) of the record indicates that it has been indexed before.

Delete The record is deleted from the index. As in the update-case it must be able to identify the record.

Please note that in both the modify- and delete- case the Zebra indexer must be able to generate a unique key that identifies the record in question (more on this below).

To administrate the Zebra retrieval system, you run the `zebraidx` program. This program supports a number of options which are preceded by a dash, and a few commands (not preceded by dash).

Both the Zebra administrative tool and the Z39.50 server share a set of index files and a global configuration file. The name of the configuration file defaults to `zebra.cfg`. The configuration file includes specifications on how to index various kinds of records and where the other configuration files are located. `zebrasrv` and `zebraidx` *must* be run in the directory where the configuration file lives unless you indicate the location of the configuration file by option `-c`.

6.1 Record Types

Indexing is a per-record process, in which either insert/modify/delete will occur. Before a record is indexed search keys are extracted from whatever might be the layout the original record (sgml,html,text, etc..). The Zebra system currently supports two fundamental types of records: structured and simple text. To specify a particular extraction process, use either the command line option `-t` or specify a `recordType` setting in the configuration file.

6.2 The Zebra Configuration File

The Zebra configuration file, read by `zebraidx` and `zebrasrv` defaults to `zebra.cfg` unless specified by `-c` option.

You can edit the configuration file with a normal text editor. parameter names and values are separated by colons in the file. Lines starting with a hash sign (`#`) are treated as comments.

If you manage different sets of records that share common characteristics, you can organize the configuration settings for each type into "groups". When `zebraidx` is run and you wish to address a given group you specify the group name with the `-g` option. In this case settings that have the group name as their prefix will be used by `zebraidx`. If no `-g` option is specified, the settings without prefix are used.

In the configuration file, the group name is placed before the option name itself, separated by a dot (.). For instance, to set the record type for group `public` to `grs.sgml` (the SGML-like format for structured records) you would write:

```
public.recordType: grs.sgml
```

To set the default value of the record type to `text` write:

```
recordType: text
```

The available configuration settings are summarized below. They will be explained further in the following sections.

group*.recordType[.*name*]: *`type` Specifies how records with the file extension *name* should be handled by the indexer. This option may also be specified as a command line option (`-t`). Note that if you do not specify a *name*, the setting applies to all files. In general, the record type specifier consists of the elements (each element separated by dot), *fundamental-type*, *file-read-type* and arguments. Currently, two fundamental types exist, `text` and `grs`.

group*.recordId: *`record-id-spec` Specifies how the records are to be identified when updated. See Section 6.3.

group*.database: *`database` Specifies the Z39.50 database name.

group*.storeKeys: *`boolean` Specifies whether key information should be saved for a given group of records. If you plan to update/delete this type of records later this should be specified as 1; otherwise it should be 0 (default), to save register space. See Section 6.5.

group*.storeData: *`boolean` Specifies whether the records should be stored internally in the Zebra system files. If you want to maintain the raw records yourself, this option should be false (0). If you want Zebra to take care of the records for you, it should be true(1).

register: *`register-location`* Specifies the location of the various register files that Zebra uses to represent your databases. See Section 6.7.

shadow: *`register-location`* Enables the *safe update* facility of Zebra, and tells the system where to place the required, temporary files. See Section 6.8.

lockDir: *`directory`* Directory in which various lock files are stored.

keyTmpDir: ***directory*** Directory in which temporary files used during zebraidx's update phase are stored.

setTmpDir: ***directory*** Specifies the directory that the server uses for temporary result sets. If not specified `/tmp` will be used.

profilePath: ***path*** Specifies a path of profile specification files. The path is composed of one or more directories separated by colon. Similar to `PATH` for UNIX systems.

modulePath: ***path*** Specifies a path of record filter modules. The path is composed of one or more directories separated by colon. Similar to `PATH` for UNIX systems. The 'make install' procedure typically puts modules in `/usr/local/lib/idzebra-2.0/modules`.

index: ***filename*** Defines the filename which holds fields structure definitions. If omitted, the file `default.idx` is read. Refer to Section 10.1 for more information.

sortmax: ***integer*** Specifies the maximum number of records that will be sorted in a result set. If the result set contains more than *integer* records, records after the limit will not be sorted. If omitted, the default value is 1,000.

staticrank: ***integer*** Enables whether static ranking is to be enabled (1) or disabled (0). If omitted, it is disabled - corresponding to a value of 0. Refer to Section 6.9.2 .

estimatehits:: ***integer*** Controls whether Zebra should calculate approximate hit counts and at which hit count it is to be enabled. A value of 0 disables approximate hit counts. For a positive value approximate hit count is enabled if it is known to be larger than *integer*.

Approximate hit counts can also be triggered by a particular attribute in a query. Refer to Section 5.3.2.5.

attset: ***filename*** Specifies the filename(s) of attribute set files for use in searching. In many configurations `bib1.att` is used, but that is not required. If Classic Explain attributes is to be used for searching, `explain.att` must be given. The path to att-files in general can be given using `profilePath` setting. See also Section 9.3.4.

memMax: ***size*** Specifies *size* of internal memory to use for the zebraidx program. The amount is given in megabytes - default is 4 (4 MB). The more memory, the faster large updates happen, up to about half the free memory available on the computer.

tempfiles: ***Yes/Auto/No*** Tells zebra if it should use temporary files when indexing. The default is Auto, in which case zebra uses temporary files only if it would need more that *memMax* megabytes of memory. This should be good for most uses.

root: ***dir*** Specifies a directory base for Zebra. All relative paths given (in profilePath, register, shadow) are based on this directory. This setting is useful if your Zebra server is running in a different directory from where `zebra.cfg` is located.

passwd: ***file*** Specifies a file with description of user accounts for Zebra. The format is similar to that known to Apache's htpasswd files and UNIX' passwd files. Non-empty lines not beginning with # are considered account lines. There is one account per-line. A line consists of fields separate by a single colon character. First field is username, second is password.

passwd.c: *file* Specifies a file with description of user accounts for Zebra. File format is similar to that used by the passwd directive except that the password are encrypted. Use Apache's htpasswd or similar for maintenance.

perm.*user*: *permstring* Specifies permissions (privilege) for a user that are allowed to access Zebra via the passwd system. There are two kinds of permissions currently: read (r) and write(w). By default users not listed in a permission directive are given the read privilege. To specify permissions for a user with no username, or Z39.50 anonymous style use `anonymous`. The permstring consists of a sequence of characters. Include character `w` for write/update access, `r` for read access and `a` to allow anonymous access through this account.

dbaccess *accessfile* Names a file which lists database subscriptions for individual users. The access file should consists of lines of the form `username: dbnames`, where dbnames is a list of database names, separated by '+'. No whitespace is allowed in the database list.

6.3 Locating Records

The default behavior of the Zebra system is to reference the records from their original location, i.e. where they were found when you run `zebraidx`. That is, when a client wishes to retrieve a record following a search operation, the files are accessed from the place where you originally put them - if you remove the files (without running `zebraidx` again, the server will return diagnostic number 14 (``System error in presenting records") to the client.

If your input files are not permanent - for example if you retrieve your records from an outside source, or if they were temporarily mounted on a CD-ROM drive, you may want Zebra to make an internal copy of them. To do this, you specify 1 (true) in the `storeData` setting. When the Z39.50 server retrieves the records they will be read from the internal file structures of the system.

6.4 Indexing with no Record IDs (Simple Indexing)

If you have a set of records that are not expected to change over time you may can build your database without record IDs. This indexing method uses less space than the other methods and is simple to use.

To use this method, you simply omit the `recordId` entry for the group of files that you index. To add a set of records you use `zebraidx` with the `update` command. The `update` command will always add all of the records that it encounters to the index - whether they have already been indexed or not. If the set of indexed files change, you should delete all of the index files, and build a new index from scratch.

Consider a system in which you have a group of text files called `simple`. That group of records should belong to a Z39.50 database called `textbase`. The following `zebra.cfg` file will suffice:

```
profilePath: /usr/local/idzebra/tab
attset: bib1.att
simple.recordType: text
simple.database: textbase
```

Since the existing records in an index can not be addressed by their IDs, it is impossible to delete or modify records when using this method.

6.5 Indexing with File Record IDs

If you have a set of files that regularly change over time: Old files are deleted, new ones are added, or existing files are modified, you can benefit from using the *file ID* indexing methodology. Examples of this type of database might include an index of WWW resources, or a USENET news spool area. Briefly speaking, the file key methodology uses the directory paths of the individual records as a unique identifier for each record. To perform indexing of a directory with file keys, again, you specify the top-level directory after the `update` command. The command will recursively traverse the directories and compare each one with whatever have been indexed before in that same directory. If a file is new (not in the previous version of the directory) it is inserted into the registers; if a file was already indexed and it has been modified since the last update, the index is also modified; if a file has been removed since the last visit, it is deleted from the index.

The resulting system is easy to administrate. To delete a record you simply have to delete the corresponding file (say, with the `rm` command). And to add records you create new files (or directories with files). For your changes to take effect in the register you must run `zebraidx update` with the same directory root again. This mode of operation requires more disk space than simpler indexing methods, but it makes it easier for you to keep the index in sync with a frequently changing set of data. If you combine this system with the *safe update* facility (see below), you never have to take your server off-line for maintenance or register updating purposes.

To enable indexing with pathname IDs, you must specify `file` as the value of `recordId` in the configuration file. In addition, you should set `storeKeys` to `1`, since the Zebra indexer must save additional information about the contents of each record in order to modify the indexes correctly at a later time.

For example, to update records of group `esdd` located below `/data1/records/` you should type:

```
$ zebraidx -g esdd update /data1/records
```

The corresponding configuration file includes:

```
esdd.recordId: file
esdd.recordType: grs.sgml
esdd.storeKeys: 1
```

Note
You cannot start out with a group of records with simple indexing (no record IDs as in the previous section) and then later enable file record Ids. Zebra must know from the first time that you index the group that the files should be indexed with file record IDs.

You cannot explicitly delete records when using this method (using the `delete` command to `zebraidx`. Instead you have to delete the files from the file system (or move them to a different location) and then run `zebraidx` with the `update` command.

6.6 Indexing with General Record IDs

When using this method you construct an (almost) arbitrary, internal record key based on the contents of the record itself and other system information. If you have a group of records that explicitly associates an

ID with each record, this method is convenient. For example, the record format may contain a title or a ID-number - unique within the group. In either case you specify the Z39.50 attribute set and use-attribute location in which this information is stored, and the system looks at that field to determine the identity of the record.

As before, the record ID is defined by the `recordId` setting in the configuration file. The value of the record ID specification consists of one or more tokens separated by whitespace. The resulting ID is represented in the index by concatenating the tokens and separating them by ASCII value (1).

There are three kinds of tokens:

Internal record info The token refers to a key that is extracted from the record. The syntax of this token is `(` *set* `,` *use* `)`, where *set* is the attribute set name *use* is the name or value of the attribute.

System variable The system variables are preceded by

```
$
```

and immediately followed by the system variable name, which may one of

group Group name.

database Current database specified.

type Record type.

Constant string A string used as part of the ID — surrounded by single- or double quotes.

For instance, the sample GILS records that come with the Zebra distribution contain a unique ID in the data tagged Control-Identifier. The data is mapped to the BIB-1 use attribute Identifier-standard (code 1007). To use this field as a record id, specify `(bib1,Identifier-standard)` as the value of the `recordId` in the configuration file. If you have other record types that uses the same field for a different purpose, you might add the record type (or group or database name) to the record id of the gils records as well, to prevent matches with other types of records. In this case the recordId might be set like this:

```
gils.recordId: $type (bib1,Identifier-standard)
```

(see Chapter 9 for details of how the mapping between elements of your records and searchable attributes is established).

As for the file record ID case described in the previous section, updating your system is simply a matter of running `zebraidx` with the `update` command. However, the update with general keys is considerably slower than with file record IDs, since all files visited must be (re)read to discover their IDs.

As you might expect, when using the general record IDs method, you can only add or modify existing records with the `update` command. If you wish to delete records, you must use the, `delete` command, with a directory as a parameter. This will remove all records that match the files below that root directory.

6.7 Register Location

Normally, the index files that form dictionaries, inverted files, record info, etc., are stored in the directory where you run `zebraidx`. If you wish to store these, possibly large, files somewhere else, you must add

the `register` entry to the `zebra.cfg` file. Furthermore, the Zebra system allows its file structures to span multiple file systems, which is useful for managing very large databases.

The value of the `register` setting is a sequence of tokens. Each token takes the form:

```
dir:size
```

The *dir* specifies a directory in which index files will be stored and the *size* specifies the maximum size of all files in that directory. The Zebra indexer system fills each directory in the order specified and use the next specified directories as needed. The *size* is an integer followed by a qualifier code, `b` for bytes, `k` for kilobytes. `M` for megabytes, `G` for gigabytes. Specifying a negative value disables the checking (it still needs the unit, use `-1b`).

For instance, if you have allocated three disks for your register, and the first disk is mounted on `/d1` and has 2GB of free space, the second, mounted on `/d2` has 3.6 GB, and the third, on which you have more space than you bother to worry about, mounted on `/d3` you could put this entry in your configuration file:

```
register: /d1:2G /d2:3600M /d3:-1b
```

Note that Zebra does not verify that the amount of space specified is actually available on the directory (file system) specified - it is your responsibility to ensure that enough space is available, and that other applications do not attempt to use the free space. In a large production system, it is recommended that you allocate one or more file system exclusively to the Zebra register files.

6.8 Safe Updating - Using Shadow Registers

6.8.1 Description

The Zebra server supports *updating* of the index structures. That is, you can add, modify, or remove records from databases managed by Zebra without rebuilding the entire index. Since this process involves modifying structured files with various references between blocks of data in the files, the update process is inherently sensitive to system crashes, or to process interruptions: Anything but a successfully completed update process will leave the register files in an unknown state, and you will essentially have no recourse but to re-index everything, or to restore the register files from a backup medium. Further, while the update process is active, users cannot be allowed to access the system, as the contents of the register files may change unpredictably.

You can solve these problems by enabling the shadow register system in Zebra. During the updating procedure, `zebraidx` will temporarily write changes to the involved files in a set of "shadow files", without modifying the files that are accessed by the active server processes. If the update procedure is interrupted by a system crash or a signal, you simply repeat the procedure - the register files have not been changed or damaged, and the partially written shadow files are automatically deleted before the new updating procedure commences.

At the end of the updating procedure (or in a separate operation, if you so desire), the system enters a "commit mode". First, any active server processes are forced to access those blocks that have been changed from the shadow files rather than from the main register files; the unmodified blocks are still accessed at their normal location (the shadow files are not a complete copy of the register files - they only contain those parts that have actually been modified). If the commit process is interrupted at any point during

the commit process, the server processes will continue to access the shadow files until you can repeat the commit procedure and complete the writing of data to the main register files. You can perform multiple update operations to the registers before you commit the changes to the system files, or you can execute the commit operation at the end of each update operation. When the commit phase has completed successfully, any running server processes are instructed to switch their operations to the new, operational register, and the temporary shadow files are deleted.

6.8.2 How to Use Shadow Register Files

The first step is to allocate space on your system for the shadow files. You do this by adding a `shadow` entry to the `zebra.cfg` file. The syntax of the `shadow` entry is exactly the same as for the `register` entry (see Section 6.7). The location of the shadow area should be *different* from the location of the main register area (if you have specified one - remember that if you provide no `register` setting, the default register area is the working directory of the server and indexing processes).

The following excerpt from a `zebra.cfg` file shows one example of a setup that configures both the main register location and the shadow file area. Note that two directories or partitions have been set aside for the shadow file area. You can specify any number of directories for each of the file areas, but remember that there should be no overlaps between the directories used for the main registers and the shadow files, respectively.

```
register: /d1:500M
shadow: /scratch1:100M /scratch2:200M
```

When shadow files are enabled, an extra command is available at the `zebraidx` command line. In order to make changes to the system take effect for the users, you'll have to submit a "commit" command after a (sequence of) update operation(s).

```
$ zebraidx update /d1/records
$ zebraidx commit
```

Or you can execute multiple updates before committing the changes:

```
$ zebraidx -g books update /d1/records  /d2/more-records
$ zebraidx -g fun update /d3/fun-records
$ zebraidx commit
```

If one of the update operations above had been interrupted, the commit operation on the last line would fail: `zebraidx` will not let you commit changes that would destroy the running register. You'll have to rerun all of the update operations since your last commit operation, before you can commit the new changes.

Similarly, if the commit operation fails, `zebraidx` will not let you start a new update operation before you have successfully repeated the commit operation. The server processes will keep accessing the shadow files rather than the (possibly damaged) blocks of the main register files until the commit operation has successfully completed.

You should be aware that update operations may take slightly longer when the shadow register system is enabled, since more file access operations are involved. Further, while the disk space required for the shadow register data is modest for a small update operation, you may prefer to disable the system if you are adding a very large number of records to an already very large database (we use the terms *large* and *modest*

very loosely here, since every application will have a different perception of size). To update the system without the use of the the shadow files, simply run `zebraidx` with the `-n` option (note that you do not have to execute the *commit* command of `zebraidx` when you temporarily disable the use of the shadow registers in this fashion. Note also that, just as when the shadow registers are not enabled, server processes will be barred from accessing the main register while the update procedure takes place.

6.9 Relevance Ranking and Sorting of Result Sets

6.9.1 Overview

The default ordering of a result set is left up to the server, which inside Zebra means sorting in ascending document ID order. This is not always the order humans want to browse the sometimes quite large hit sets. Ranking and sorting comes to the rescue.

In cases where a good presentation ordering can be computed at indexing time, we can use a fixed `static ranking` scheme, which is provided for the `alvis` indexing filter. This defines a fixed ordering of hit lists, independently of the query issued.

There are cases, however, where relevance of hit set documents is highly dependent on the query processed. Simply put, `dynamic relevance ranking` sorts a set of retrieved records such that those most likely to be relevant to your request are retrieved first. Internally, Zebra retrieves all documents that satisfy your query, and re-orders the hit list to arrange them based on a measurement of similarity between your query and the content of each record.

Finally, there are situations where hit sets of documents should be `sorted` during query time according to the lexicographical ordering of certain sort indexes created at indexing time.

6.9.2 Static Ranking

Zebra uses internally inverted indexes to look up term frequencies in documents. Multiple queries from different indexes can be combined by the binary boolean operations `AND`, `OR` and/or `NOT` (which is in fact a binary `AND NOT` operation). To ensure fast query execution speed, all indexes have to be sorted in the same order.

The indexes are normally sorted according to document `ID` in ascending order, and any query which does not invoke a special re-ranking function will therefore retrieve the result set in document `ID` order.

If one defines the

```
staticrank: 1
```

directive in the main core Zebra configuration file, the internal document keys used for ordering are augmented by a preceding integer, which contains the static rank of a given document, and the index lists are ordered first by ascending static rank, then by ascending document `ID`. Zero is the ``best'' rank, as it occurs at the beginning of the list; higher numbers represent worse scores.

The experimental `alvis` filter provides a directive to fetch static rank information out of the indexed XML records, thus making *all* hit sets ordered after *ascending* static rank, and for those doc's which have the same static rank, ordered after *ascending* doc `ID`. See Chapter 8 for the gory details.

6.9.3 Dynamic Ranking

In order to fiddle with the static rank order, it is necessary to invoke additional re-ranking/re-ordering using dynamic ranking or score functions. These functions return positive integer scores, where *highest* score is ``best''; hit sets are sorted according to *descending* scores (in contrary to the index lists which are sorted according to ascending rank number and document ID).

Dynamic ranking is enabled by a directive like one of the following in the zebra configuration file (use only one of these a time!):

```
rank: rank-1         # default TDF-IDF like
rank: rank-static    # dummy do-nothing
```

Dynamic ranking is done at query time rather than indexing time (this is why we call it ``dynamic ranking'' in the first place ...) It is invoked by adding the BIB-1 relation attribute with value ``relevance'' to the PQF query (that is, `@attr 2=102`, see also The BIB-1 Attribute Set Semantics, also in HTML). To find all articles with the word `Eoraptor` in the title, and present them relevance ranked, issue the PQF query:

```
@attr 2=102 @attr 1=4 Eoraptor
```

6.9.3.1 Dynamically ranking using PQF queries with the 'rank-1' algorithm

The default `rank-1` ranking module implements a TF/IDF (Term Frequecy over Inverse Document Frequency) like algorithm. In contrast to the usual definition of TF/IDF algorithms, which only considers searching in one full-text index, this one works on multiple indexes at the same time. More precisely, Zebra does boolean queries and searches in specific addressed indexes (there are inverted indexes pointing from terms in the dictionary to documents and term positions inside documents). It works like this:

Query Components First, the boolean query is dismantled into its principal components, i.e. atomic queries where one term is looked up in one index. For example, the query

```
@attr 2=102 @and @attr 1=1010 Utah @attr 1=1018 Springer
```

is a boolean AND between the atomic parts

```
@attr 2=102 @attr 1=1010 Utah
```

and

```
@attr 2=102 @attr 1=1018 Springer
```

which gets processed each for itself.

Atomic hit lists Second, for each atomic query, the hit list of documents is computed.

In this example, two hit lists for each index `@attr 1=1010` and `@attr 1=1018` are computed.

Atomic scores Third, each document in the hit list is assigned a score (_if_ ranking is enabled and requested in the query) using a TF/IDF scheme.

In this example, both atomic parts of the query assign the magic `@attr 2=102` relevance attribute, and are to be used in the relevance ranking functions.

It is possible to apply dynamic ranking on only parts of the PQF query:

```
@and @attr 2=102 @attr 1=1010 Utah @attr 1=1018 Springer
```

searches for all documents which have the term 'Utah' on the body of text, and which have the term 'Springer' in the publisher field, and sort them in the order of the relevance ranking made on the body-of-text index only.

Hit list merging Fourth, the atomic hit lists are merged according to the boolean conditions to a final hit list of documents to be returned.

This step is always performed, independently of the fact that dynamic ranking is enabled or not.

Document score computation Fifth, the total score of a document is computed as a linear combination of the atomic scores of the atomic hit lists

Ranking weights may be used to pass a value to a ranking algorithm, using the non-standard BIB-1 attribute type 9. This allows one branch of a query to use one value while another branch uses a different one. For example, we can search for `utah` in the `@attr 1=4` index with weight 30, as well as in the `@attr 1=1010` index with weight 20:

```
@attr 2=102 @or @attr 9=30 @attr 1=4 utah @attr 9=20 @attr ↩
   1=1010 city
```

The default weight is sqrt(1000) ~ 34 , as the Z39.50 standard prescribes that the top score is 1000 and the bottom score is 0, encoded in integers.

Warning
The ranking-weight feature is experimental. It may change in future releases of zebra.

Re-sorting of hit list Finally, the final hit list is re-ordered according to scores.

The `rank-1` algorithm does not use the static rank information in the list keys, and will produce the same ordering with or without static ranking enabled.

Warning
`Dynamic ranking` is not compatible with `estimated hit sizes`, as all documents in a hit set must be accessed to compute the correct placing in a ranking sorted list. Therefore the use attribute setting `@attr 2=102` clashes with `@attr 9=integer`.

6.9.3.2 Dynamically ranking CQL queries

Dynamic ranking can be enabled during sever side CQL query expansion by adding `@attr 2=102` chunks to the CQL config file. For example

```
relationModifier.relevant    = 2=102
```

invokes dynamic ranking each time a CQL query of the form

```
Z> querytype cql
Z> f alvis.text =/relevant house
```

is issued. Dynamic ranking can also be automatically used on specific CQL indexes by (for example) setting

```
index.alvis.text                          = 1=text 2=102
```

which then invokes dynamic ranking each time a CQL query of the form

```
Z> querytype cql
Z> f alvis.text = house
```

is issued.

6.9.4 Sorting

Zebra sorts efficiently using special sorting indexes (type=s; so each sortable index must be known at indexing time, specified in the configuration of record indexing. For example, to enable sorting according to the BIB-1 Date/time-added-to-db field, one could add the line

```
 xelm /*/@created              Date/time-added-to-db:s
```

to any .abs record-indexing configuration file. Similarly, one could add an indexing element of the form

```
<z:index name="date-modified" type="s">
 <xsl:value-of select="some/xpath"/>
</z:index>
```

to any alvis-filter indexing stylesheet.

Indexing can be specified at searching time using a query term carrying the non-standard BIB-1 attribute-type 7. This removes the need to send a Z39.50 Sort Request separately, and can dramatically improve latency when the client and server are on separate networks. The sorting part of the query is separate from the rest of the query - the actual search specification - and must be combined with it using OR.

A sorting subquery needs two attributes: an index (such as a BIB-1 type-1 attribute) specifying which index to sort on, and a type-7 attribute whose value is be 1 for ascending sorting, or 2 for descending. The term associated with the sorting attribute is the priority of the sort key, where 0 specifies the primary sort key, 1 the secondary sort key, and so on.

For example, a search for water, sort by title (ascending), is expressed by the PQF query

```
@or @attr 1=1016 water @attr 7=1 @attr 1=4 0
```

whereas a search for water, sort by title ascending, then date descending would be

```
@or @or @attr 1=1016 water @attr 7=1 @attr 1=4 0 @attr 7=2 @attr 1=30  ↩
   1
```

Notice the fundamental differences between dynamic ranking and sorting: there can be only one ranking function defined and configured; but multiple sorting indexes can be specified dynamically at search time. Ranking does not need to use specific indexes, so dynamic ranking can be enabled and disabled without re-indexing; whereas, sorting indexes need to be defined before indexing.

6.10 Extended Services: Remote Insert, Update and Delete

Note
Extended services are only supported when accessing the Zebra server using the Z39.50 protocol. The SRU protocol does not support extended services.

The extended services are not enabled by default in zebra - due to the fact that they modify the system. Zebra can be configured to allow anybody to search, and to allow only updates for a particular admin user in the main zebra configuration file `zebra.cfg`. For user `admin`, you could use:

```
perm.anonymous: r
perm.admin: rw
passwd: passwordfile
```

And in the password file `passwordfile`, you have to specify users and encrypted passwords as colon separated strings. Use a tool like `htpasswd` to maintain the encrypted passwords.

```
admin:secret
```

It is essential to configure Zebra to store records internally, and to support modifications and deletion of records:

```
storeData: 1
storeKeys: 1
```

The general record type should be set to any record filter which is able to parse XML records, you may use any of the two declarations (but not both simultaneously!)

```
recordType: dom.filter_dom_conf.xml
# recordType: grs.xml
```

Notice the difference to the specific instructions

```
recordType.xml: dom.filter_dom_conf.xml
# recordType.xml: grs.xml
```

which only work when indexing XML files from the filesystem using the `*.xml` naming convention.

To enable transaction safe shadow indexing, which is extra important for this kind of operation, set

```
shadow: directoryname: size (e.g. 1000M)
```

See Section 6.2 for additional information on these configuration options.

Note
It is not possible to carry information about record types or similar to Zebra when using extended services, due to limitations of the Z39.50 protocol. Therefore, indexing filters can not be chosen on a per-record basis. One and only one general XML indexing filter must be defined.

6.10.1 Extended services in the Z39.50 protocol

The Z39.50 standard allows servers to accept special binary *extended services* protocol packages, which may be used to insert, update and delete records into servers. These carry control and update information to the servers, which are encoded in seven package fields:

Parameter	Value	Notes
`type`	`'update'`	Must be set to trigger extended services
`action`	`string`	Extended service action type with one of four possible values: `recordInsert`, `recordReplace`, `recordDelete`, and `specialUpdate`
`record`	`XML string`	An XML formatted string containing the record
`syntax`	`'xml'`	XML/SUTRS/MARC. GRS-1 not supported. The default filter (record type) as given by recordType in zebra.cfg is used to parse the record.
`recordIdOpaque`	`string`	Optional client-supplied, opaque record identifier used under insert operations.
`recordIdNumber`	`positive number`	Zebra's internal system number, not allowed for `recordInsert` or `specialUpdate` actions which result in fresh record inserts.
`databaseName`	`database identifier`	The name of the database to which the extended services should be applied.

Table 6.1: Extended services Z39.50 Package Fields

The `action` parameter can be any of `recordInsert` (will fail if the record already exists), `recordReplace` (will fail if the record does not exist), `recordDelete` (will fail if the record does not exist), and `specialUpdate` (will insert or update the record as needed, record deletion is not possible).

During all actions, the usual rules for internal record ID generation apply, unless an optional `recordIdNumber` Zebra internal ID or a `recordIdOpaque` string identifier is assigned. The default ID generation is configured using the `recordId:` from `zebra.cfg`. See Section 6.2.

Setting of the `recordIdNumber` parameter, which must be an existing Zebra internal system ID number, is not allowed during any `recordInsert` or `specialUpdate` action resulting in fresh record inserts.

When retrieving existing records indexed with GRS-1 indexing filters, the Zebra internal ID number is returned in the field `/*/id:idzebra/localnumber` in the namespace `xmlns:id="http://www.indexdata.dk/zebra/"`, where it can be picked up for later record updates or deletes.

A new element set for retrieval of internal record data has been added, which can be used to access minimal records containing only the `recordIdNumber` Zebra internal ID, or the `recordIdOpaque` string identifier. This works for any indexing filter used. See Section 4.4.

The `recordIdOpaque` string parameter is an client-supplied, opaque record identifier, which may be used under insert, update and delete operations. The client software is responsible for assigning these to records. This identifier will replace zebra's own automagic identifier generation with a unique mapping from `recordIdOpaque` to the Zebra internal `recordIdNumber`. *The opaque `recordIdOpaque` string identifiers are not visible in retrieval records, nor are searchable, so the value of this parameter is questionable. It serves mostly as a convenient mapping from application domain string identifiers to Zebra internal ID's.*

6.10.2 Extended services from yaz-client

We can now start a yaz-client admin session and create a database:

```
$ yaz-client localhost:9999 -u admin/secret
Z> adm-create
```

Now the `Default` database was created, we can insert an XML file (esdd0006.grs from example/gils/records) and index it:

```
Z> update insert id1234 esdd0006.grs
```

The 3rd parameter - `id1234` here - is the `recordIdOpaque` package field.

Actually, we should have a way to specify "no opaque record id" for yaz-client's update command.. We'll fix that.

The newly inserted record can be searched as usual:

```
Z> f utah
Sent searchRequest.
Received SearchResponse.
Search was a success.
Number of hits: 1, setno 1
SearchResult-1: term=utah cnt=1
records returned: 0
Elapsed: 0.014179
```

Let's delete the beast, using the same `recordIdOpaque` string parameter:

```
Z> update delete id1234
No last record (update ignored)
Z> update delete 1 esdd0006.grs
Got extended services response
Status: done
Elapsed: 0.072441
Z> f utah
Sent searchRequest.
Received SearchResponse.
Search was a success.
Number of hits: 0, setno 2
SearchResult-1: term=utah cnt=0
records returned: 0
Elapsed: 0.013610
```

If shadow register is enabled in your `zebra.cfg`, you must run the adm-commit command

```
Z> adm-commit
```

after each update session in order write your changes from the shadow to the life register space.

6.10.3 Extended services from yaz-php

Extended services are also available from the YAZ PHP client layer. An example of an YAZ-PHP extended service transaction is given here:

```
$record = '<record><title>A fine specimen of a record</title></record ↩
   >';

$options = array('action' => 'recordInsert',
                 'syntax' => 'xml',
                 'record' => $record,
                 'databaseName' => 'mydatabase'
                );

yaz_es($yaz, 'update', $options);
yaz_es($yaz, 'commit', array());
yaz_wait();

if ($error = yaz_error($yaz))
  echo "$error";
```

6.10.4 Extended services debugging guide

When debugging ES over PHP we recommend the following order of tests:

- Make sure you have a nice record on your filesystem, which you can index from the filesystem by use of the zebraidx command. Do it exactly as you planned, using one of the GRS-1 filters, or the DOMXML filter. When this works, proceed.
- Check that your server setup is OK before you even coded one single line PHP using ES. Take the same record form the file system, and send as ES via `yaz-client` like described in Section 6.10.2, and remember the `-a` option which tells you what goes over the wire! Notice also the section on permissions: try

  ```
  perm.anonymous: rw
  ```

  in `zebra.cfg` to make sure you do not run into permission problems (but never expose such an insecure setup on the internet!!!). Then, make sure to set the general `recordType` instruction, pointing correctly to the GRS-1 filters, or the DOMXML filters.
- If you insist on using the `sysno` in the `recordIdNumber` setting, please make sure you do only updates and deletes. Zebra's internal system number is not allowed for `recordInsert` or `specialUpdate` actions which result in fresh record inserts.
- If `shadow register` is enabled in your `zebra.cfg`, you must remember running the

  ```
  Z> adm-commit
  ```

  command as well.
- If this works, then proceed to do the same thing in your PHP script.

Chapter 7

DOM XML Record Model and Filter Module

The record model described in this chapter applies to the fundamental, structured XML record type `DOM`, introduced in Section 4.2.5.1. The DOM XML record model is experimental, and its inner workings might change in future releases of the Zebra Information Server.

7.1 DOM Record Filter Architecture

The DOM XML filter uses a standard DOM XML structure as internal data model, and can therefore parse, index, and display any XML document type. It is well suited to work on standardized XML-based formats such as Dublin Core, MODS, METS, MARCXML, OAI-PMH, RSS, and performs equally well on any other non-standard XML format.

A parser for binary MARC records based on the ISO2709 library standard is provided, it transforms these to the internal MARCXML DOM representation. Other binary document parsers are planned to follow.

The DOM filter architecture consists of four different pipelines, each being a chain of arbitrarily many successive XSLT transformations of the internal DOM XML representations of documents.

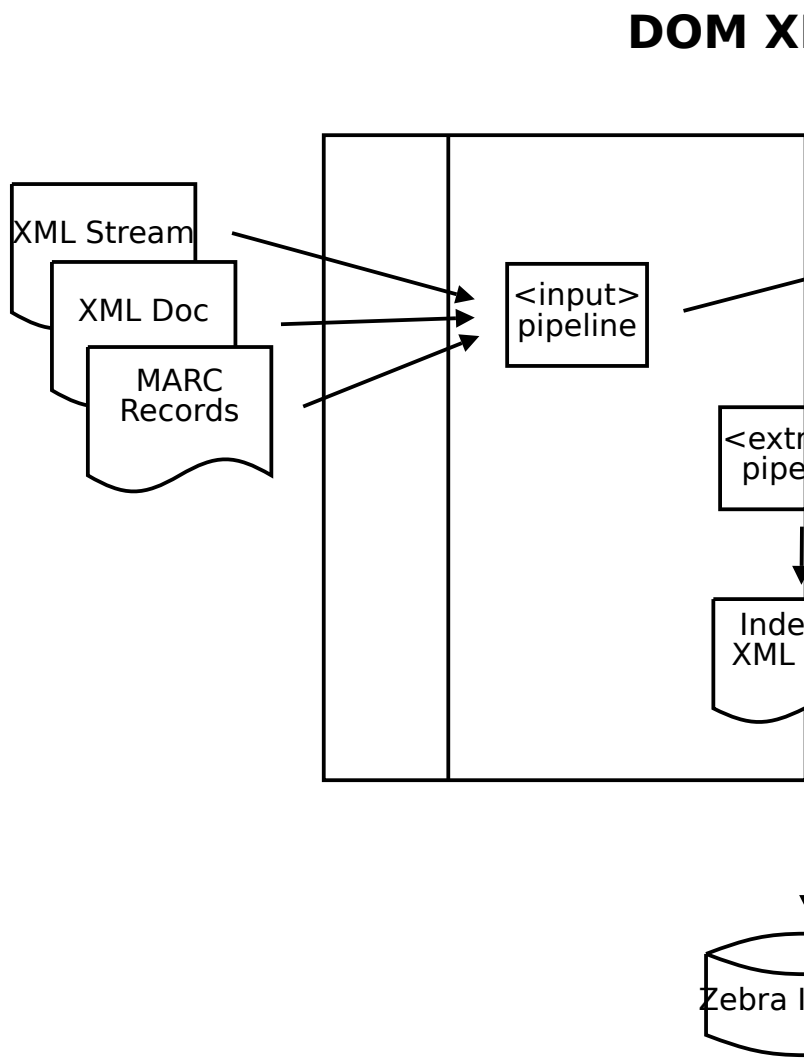


Figure 7.1: DOM XML filter architecture

The DOM XML filter pipelines use XSLT (and if supported on your platform, even EXSLT), it brings thus full XPATH support to the indexing, storage and display rules of not only XML documents, but also binary MARC records.

7.2 DOM XML filter pipeline configuration

The experimental, loadable DOM XML/XSLT filter module `mod-dom.so` is invoked by the `zebra.cfg` configuration statement

```
recordtype.xml: dom.db/filter_dom_conf.xml
```

In this example the DOM XML filter is configured to work on all data files with suffix `*.xml`, where the configuration file is found in the path `db/filter_dom_conf.xml`.

Name	When	Description	Input	Output
`input`	first	input parsing and initial transformations to common XML format	Input raw XML record buffers, XML streams and binary MARC buffers	Common XML DOM
`extract`	second	indexing term extraction transformations	Common XML DOM	Indexing XML DOM
`store`	second	transformations before internal document storage	Common XML DOM	Storage XML DOM
`retrieve`	third	multiple document retrieve transformations from storage to different output formats are possible	Storage XML DOM	Output XML syntax in requested formats

Table 7.1: DOM XML filter pipelines overview

The DOM XSLT filter configuration file must be valid XML. It might look like this:

```
<?xml version="1.0" encoding="UTF8"?>
<dom xmlns="http://indexdata.com/zebra-2.0">
  <input>
    <xmlreader level="1"/>
    <!-- <marc inputcharset="marc-8"/> -->
  </input>
  <extract>
     <xslt stylesheet="common2index.xsl"/>
  </extract>
  <store>
     <xslt stylesheet="common2store.xsl"/>
  </store>
  <retrieve name="dc">
    <xslt stylesheet="store2dc.xsl"/>
  </retrieve>
  <retrieve name="mods">
    <xslt stylesheet="store2mods.xsl"/>
  </retrieve>
</dom>
```

The root XML element `<dom>` and all other DOM XML filter elements are residing in the namespace `xmlns="http://indexdata.com/zebra-2.0"`.

All pipeline definition elements - i.e. the <input>, <extract>, <store>, and <retrieve> elements - are optional. Missing pipeline definitions are just interpreted do-nothing identity pipelines.

All pipeline definition elements may contain zero or more <xslt stylesheet="path/file.xsl"/> XSLT transformation instructions, which are performed sequentially from top to bottom. The paths in the stylesheet attributes are relative to zebras working directory, or absolute to the file system root.

7.2.1 Input pipeline

The <input> pipeline definition element may contain either one XML Reader definition <xmlreader level="1"/>, used to split an XML collection input stream into individual XML DOM documents at the prescribed element level, or one MARC binary parsing instruction <marc inputcharset="marc-8"/>, which defines a conversion to MARCXML format DOM trees. The allowed values of the inputcharset attribute depend on your local iconv set-up.

Both input parsers deliver individual DOM XML documents to the following chain of zero or more <xslt stylesheet="path/file.xsl"/> XSLT transformations. At the end of this pipeline, the documents are in the common format, used to feed both the <extract> and <store> pipelines.

7.2.2 Extract pipeline

The <extract> pipeline takes documents from any common DOM XML format to the Zebra specific indexing DOM XML format. It may consist of zero ore more <xslt stylesheet="path/file.xsl"/> XSLT transformations, and the outcome is handled to the Zebra core to drive the process of building the inverted indexes. See Section 7.2.5 for details.

7.2.3 Store pipeline

The <store> pipeline takes documents from any common DOMXML format to the Zebra specific storage DOMXML format. It may consist of zero ore more <xslt stylesheet="path/file.xsl"/>XSLT transformations, and the outcome is handled to the Zebra core for deposition into the internal storage system.

7.2.4 Retrieve pipeline

Finally, there may be one or more <retrieve> pipeline definitions, each of them again consisting of zero or more <xslt stylesheet="path/file.xsl"/> XSLT transformations. These are used for document presentation after search, and take the internal storage DOM XML to the requested output formats during record present requests.

The possible multiple <retrieve> pipeline definitions are distinguished by their unique name attributes, these are the literal schema or element set names used in SRW, SRU and Z39.50 protocol queries.

7.2.5 Canonical Indexing Format

DOM XML indexing comes in two flavors: pure processing-instruction governed plain XML documents, and - very similar to the Alvis filter indexing format - XML documents containing XML `<record>` and `<index>` instructions from the magic namespace `xmlns:z="http://indexdata.com/zebra--2.0"`.

7.2.5.1 Processing-instruction governed indexing format

The output of the processing instruction driven indexing XSLT stylesheets must contain processing instructions named `zebra-2.0`. The output of the XSLT indexing transformation is then parsed using DOM methods, and the contained instructions are performed on the *elements and their subtrees directly following the processing instructions*.

For example, the output of the command

```
xsltproc dom-index-pi.xsl marc-one.xml
```

might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?zebra-2.0 record id=11224466 rank=42?>
<record>
  <?zebra-2.0 index control:0?>
  <control>11224466</control>
  <?zebra-2.0 index any:w title:w title:p title:s?>
  <title>How to program a computer</title>
</record>
```

7.2.5.2 Magic element governed indexing format

The output of the indexing XSLT stylesheets must contain certain elements in the magic `xmlns:z="http://indexdata.com/zebra-2.0"` namespace. The output of the XSLT indexing transformation is then parsed using DOM methods, and the contained instructions are performed on the *magic elements and their subtrees*.

For example, the output of the command

```
xsltproc dom-index-element.xsl marc-one.xml
```

might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<z:record xmlns:z="http://indexdata.com/zebra-2.0"
          z:id="11224466" z:rank="42">
    <z:index name="control:0">11224466</z:index>
    <z:index name="any:w title:w title:p title:s">
```

```
                    How to program a computer</z:index>
    </z:record>
```

7.2.5.3 Semantics of the indexing formats

Both indexing formats are defined with equal semantics and behavior in mind:

- Zebra specific instructions are either processing instructions named `zebra-2.0` or elements contained in the namespace `xmlns:z="http://indexdata.com/zebra-2.0"`.
- There must be exactly one `record` instruction, which sets the scope for the following, possibly nested `index` instructions.
- The unique `record` instruction may have additional attributes `id`, `rank` and `type`. Attribute `id` is the value of the opaque ID and may be any string not containing the whitespace character `' '`. The `rank` attribute value must be a non-negative integer. See Section 6.9 . The `type` attribute specifies how the record is to be treated. The following values may be given for `type`:

  `insert` The record is inserted. If the record already exists, it is skipped (i.e. not replaced).

  `replace` The record is replaced. If the record does not already exist, it is skipped (i.e. not inserted).

  `delete` The record is deleted. If the record does not already exist, it is skipped (i.e. nothing is deleted).

  `update` The record is inserted or replaced depending on whether the record exists or not. This is the default behavior but may be effectively changed by "outside" the scope of the DOM filter by zebraidx commands or extended services updates.

  Note that the value of `type` is only used to determine the action if and only if the Zebra indexer is running in "update" mode (i.e zebraidx update) or if the specialUpdate action of the Extended Service Update is used. For this reason a specialUpdate may end up deleting records!

- Multiple and possible nested `index` instructions must contain at least one `indexname:indextype` pair, and may contain multiple such pairs separated by the whitespace character `' '`. In each index pair, the name and the type of the index is separated by a colon character `':'`.
- Any index name consisting of ASCII letters, and following the standard Zebra rules will do, see Section 5.3.5.1.
- Index types are restricted to the values defined in the standard configuration file `default.idx`, see Section 5.2.3 and Chapter 10 for details.
- DOM input documents which are not resulting in both one unique valid `record` instruction and one or more valid `index` instructions can not be searched and found. Therefore, invalid document processing is aborted, and any content of the `<extract>` and `<store>` pipelines is discarded. A warning is issued in the logs.

The examples work as follows: From the original XML file `marc-one.xml` (or from the XML record DOM of the same form coming from an `<input>` pipeline), the indexing pipeline `<extract>` produces an indexing XML record, which is defined by the `record` instruction Zebra uses the content of `z:id="11224466"` or `id=11224466` as internal record ID, and - in case static ranking is set - the content of `rank=42` or `z:rank="42"` as static rank.

In these examples, the following literal indexes are constructed:

```
any:w
control:0
title:w
title:p
title:s
```

where the indexing type is defined after the literal `':'` character. Any value from the standard configuration file `default.idx` will do. Finally, any `text()` node content recursively contained inside the `<z:index>` element, or any element following a `index` processing instruction, will be filtered through the appropriate char map for character normalization, and will be inserted in the named indexes.

Finally, this example configuration can be queried using PQF queries, either transported by Z39.50, (here using a yaz-client)

```
Z> open localhost:9999
Z> elem dc
Z> form xml
Z>
Z> find @attr 1=control @attr 4=3 11224466
Z> scan @attr 1=control @attr 4=3 ""
Z>
Z> find @attr 1=title program
Z> scan @attr 1=title ""
Z>
Z> find @attr 1=title @attr 4=2 "How to program a computer"
Z> scan @attr 1=title @attr 4=2 ""
```

or the proprietary extensions `x-pquery` and `x-pScanClause` to SRU, and SRW

```
http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery= ↩
   @attr 1=title program
http://localhost:9999/?version=1.1&operation=scan&x-pScanClause=@attr ↩
    1=title ""
```

See the section called "The SRU Server" for more information on SRU/SRW configuration, and the section called "YAZ server virtual hosts" or the YAZ CQL section for the details or the YAZ frontend server.

Notice that there are no `*.abs`, `*.est`, `*.map`, or other GRS-1 filter configuration files involves in this process, and that the literal index names are used during search and retrieval.

In case that we want to support the usual `bib-1` Z39.50 numeric access points, it is a good idea to choose string index names defined in the default configuration file `tab/bib1.att`, see Section 9.3.4

7.3 DOM Record Model Configuration

7.3.1 DOM Indexing Configuration

As mentioned above, there can be only one indexing pipeline, and configuration of the indexing process is a synonym of writing an XSLT stylesheet which produces XML output containing the magic processing instructions or elements discussed in Section 7.2.5. Obviously, there are million of different ways to accomplish this task, and some comments and code snippets are in order to enlighten the wary.

Stylesheets can be written in the *pull* or the *push* style: *pull* means that the output XML structure is taken as starting point of the internal structure of the XSLT stylesheet, and portions of the input XML are *pulled* out and inserted into the right spots of the output XML structure. On the other side, *push* XSLT stylesheets are recursively calling their template definitions, a process which is commanded by the input XML structure, and is triggered to produce some output XML whenever some special conditions in the input stylesheets are met. The *pull* type is well-suited for input XML with strong and well-defined structure and semantics, like the following OAI indexing example, whereas the *push* type might be the only possible way to sort out deeply recursive input XML formats.

A *pull* stylesheet example used to index OAI harvested records could use some of the following template definitions:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:z="http://indexdata.com/zebra-2.0"
 xmlns:oai="http://www.openarchives.org/&acro.oai;/2.0/"
 xmlns:oai_dc="http://www.openarchives.org/&acro.oai;/2.0/oai_dc/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 version="1.0">

 <!-- Example pull and magic element style Zebra indexing -->
 <xsl:output indent="yes" method="xml" version="1.0" encoding="UTF ↩
    -8"/>

  <!-- disable all default text node output -->
  <xsl:template match="text()"/>

  <!-- disable all default recursive element node transversal -->
  <xsl:template match="node()"/>

   <!-- match only on oai xml record root -->
   <xsl:template match="/">
    <z:record z:id="{normalize-space(oai:record/oai:header/oai: ↩
       identifier)}">
     <!-- you may use z:rank="{some XSLT; function here}" -->

     <!-- explicetly calling defined templates -->
     <xsl:apply-templates/>
    </z:record>
   </xsl:template>
```

```
    <!-- OAI indexing templates -->
    <xsl:template match="oai:record/oai:header/oai:identifier">
     <z:index name="oai_identifier:0">
      <xsl:value-of select="."/>
     </z:index>
    </xsl:template>

    <!-- etc, etc -->

    <!-- DC specific indexing templates -->
    <xsl:template match="oai:record/oai:metadata/oai_dc:dc/dc:title">
     <z:index name="dc_any:w dc_title:w dc_title:p dc_title:s ">
      <xsl:value-of select="."/>
     </z:index>
    </xsl:template>

    <!-- etc, etc -->

</xsl:stylesheet>
```

7.3.2 DOM Indexing MARCXML

The DOM filter allows indexing of both binary MARC records and MARCXML records, depending on its configuration. A typical MARCXML record might look like this:

```
<record xmlns="http://www.loc.gov/MARC21/slim">
 <rank>42</rank>
 <leader>00366nam  22001698a 4500</leader>
 <controlfield tag="001">   11224466   </controlfield>
 <controlfield tag="003">DLC  </controlfield>
 <controlfield tag="005">00000000000000.0  </controlfield>
 <controlfield tag="008">910710c19910701nju           00010 eng    </ ↩
    controlfield>
 <datafield tag="010" ind1=" " ind2=" ">
   <subfield code="a">   11224466 </subfield>
 </datafield>
 <datafield tag="040" ind1=" " ind2=" ">
   <subfield code="a">DLC</subfield>
   <subfield code="c">DLC</subfield>
 </datafield>
 <datafield tag="050" ind1="0" ind2="0">
   <subfield code="a">123-xyz</subfield>
 </datafield>
 <datafield tag="100" ind1="1" ind2="0">
   <subfield code="a">Jack Collins</subfield>
 </datafield>
 <datafield tag="245" ind1="1" ind2="0">
```

```
    <subfield code="a">How to program a computer</subfield>
  </datafield>
  <datafield tag="260" ind1="1" ind2=" ">
    <subfield code="a">Penguin</subfield>
  </datafield>
  <datafield tag="263" ind1=" " ind2=" ">
    <subfield code="a">8710</subfield>
  </datafield>
  <datafield tag="300" ind1=" " ind2=" ">
    <subfield code="a">p. cm.</subfield>
  </datafield>
 </record>
```

It is easily possible to make string manipulation in the DOM filter. For example, if you want to drop some leading articles in the indexing of sort fields, you might want to pick out the MARCXML indicator attributes to chop of leading substrings. If the above XML example would have an indicator `ind2="8"` in the title field `245`, i.e.

```
  <datafield tag="245" ind1="1" ind2="8">
    <subfield code="a">How to program a computer</subfield>
  </datafield>
```

one could write a template taking into account this information to chop the first `8` characters from the sorting index `title:s` like this:

```
 <xsl:template match="m:datafield[@tag='245']">
   <xsl:variable name="chop">
     <xsl:choose>
       <xsl:when test="not(number(@ind2))">0</xsl:when>
       <xsl:otherwise><xsl:value-of select="number(@ind2)"/></xsl: ↩
          otherwise>
     </xsl:choose>
   </xsl:variable>

   <z:index name="title:w title:p any:w">
      <xsl:value-of select="m:subfield[@code='a']"/>
   </z:index>

   <z:index name="title:s">
     <xsl:value-of select="substring(m:subfield[@code='a'], $chop)"/>
   </z:index>

 </xsl:template>
```

The output of the above MARCXML and XSLT excerpt would then be:

```
  <z:index name="title:w title:p any:w">How to program a computer</z: ↩
     index>
  <z:index name="title:s">program a computer</z:index>
```

and the record would be sorted in the title index under 'P', not 'H'.

7.3.3 DOM Indexing Wizardry

The names and types of the indexes can be defined in the indexing XSLT stylesheet *dynamically according to content in the original XML records*, which has opportunities for great power and wizardry as well as grande disaster.

The following excerpt of a *push* stylesheet *might* be a good idea according to your strict control of the XML input format (due to rigorous checking against well-defined and tight RelaxNG or XML Schema's, for example):

```
<xsl:template name="element-name-indexes">
 <z:index name="{name()}:w">
  <xsl:value-of select="'1'"/>
 </z:index>
</xsl:template>
```

This template creates indexes which have the name of the working node of any input XML file, and assigns a '1' to the index. The example query `find @attr 1=xyz 1` finds all files which contain at least one `xyz` XML element. In case you can not control which element names the input files contain, you might ask for disaster and bad karma using this technique.

One variation over the theme *dynamically created indexes* will definitely be unwise:

```
<!-- match on oai xml record root -->
<xsl:template match="/">
 <z:record>

  <!-- create dynamic index name from input content -->
  <xsl:variable name="dynamic_content">
   <xsl:value-of select="oai:record/oai:header/oai:identifier"/>
  </xsl:variable>

  <!-- create zillions of indexes with unknown names -->
  <z:index name="{$dynamic_content}:w">
   <xsl:value-of select="oai:record/oai:metadata/oai_dc:dc"/>
  </z:index>
 </z:record>

</xsl:template>
```

Don't be tempted to play too smart tricks with the power of XSLT, the above example will create zillions of indexes with unpredictable names, resulting in severe Zebra index pollution..

7.3.4 Debuggig DOM Filter Configurations

It can be very hard to debug a DOM filter setup due to the many successive MARC syntax translations, XML stream splitting and XSLT transformations involved. As an aid, you have always the power of the `-s` command line switch to the `zebraidz` indexing command at your hand:

```
zebraidx -s -c zebra.cfg update some_record_stream.xml
```

This command line simulates indexing and dumps a lot of debug information in the logs, telling exactly which transformations have been applied, how the documents look like after each transformation, and which record ids and terms are send to the indexer.

Chapter 8

ALVIS XML Record Model and Filter Module

Warning
The functionality of this record model has been improved and replaced by the DOM XML record model, see Chapter 7. The Alvis XML record model is considered obsolete, and will eventually be removed from future releases of the Zebra software.

The record model described in this chapter applies to the fundamental, structured XML record type `alvis`, introduced in Section 4.2.5.2.

This filter has been developed under the ALVIS project funded by the European Community under the "Information Society Technologies" Program (2002-2006).

8.1 ALVIS Record Filter

The experimental, loadable Alvis XML/XSLT filter module `mod-alvis.so` is packaged in the GNU/Debian package `libidzebra1.4-mod-alvis`. It is invoked by the `zebra.cfg` configuration statement

```
recordtype.xml: alvis.db/filter_alvis_conf.xml
```

In this example on all data files with suffix `*.xml`, where the Alvis XSLT filter configuration file is found in the path `db/filter_alvis_conf.xml`.

The Alvis XSLT filter configuration file must be valid XML. It might look like this (This example is used for indexing and display of OAI harvested records):

```
<?xml version="1.0" encoding="UTF-8"?>
  <schemaInfo>
    <schema name="identity" stylesheet="xsl/identity.xsl" />
    <schema name="index" identifier="http://indexdata.dk/zebra/xslt/1"
        stylesheet="xsl/oai2index.xsl" />
    <schema name="dc" stylesheet="xsl/oai2dc.xsl" />
    <!-- use split level 2 when indexing whole OAI Record lists -->
    <split level="2"/>
  </schemaInfo>
```

All named stylesheets defined inside `schema` element tags are for presentation after search, including the indexing stylesheet (which is a great debugging help). The names defined in the `name` attributes must be unique, these are the literal `schema` or `element set` names used in SRW, SRU and Z39.50 protocol queries. The paths in the `stylesheet` attributes are relative to zebras working directory, or absolute to file system root.

The `<split level="2"/>` decides where the XML Reader shall split the collections of records into individual records, which then are loaded into DOM, and have the indexing XSLT stylesheet applied.

There must be exactly one indexing XSLT stylesheet, which is defined by the magic attribute `identifier="http://indexdata.dk/zebra/xslt/1"`.

8.1.1 ALVIS Internal Record Representation

When indexing, an XML Reader is invoked to split the input files into suitable record XML pieces. Each record piece is then transformed to an XML DOM structure, which is essentially the record model. Only XSLT transformations can be applied during index, search and retrieval. Consequently, output formats are restricted to whatever XSLT can deliver from the record XML structure, be it other XML formats, HTML, or plain text. In case you have `libxslt1` running with EXSLT support, you can use this functionality inside the Alvis filter configuration XSLT stylesheets.

8.1.2 ALVIS Canonical Indexing Format

The output of the indexing XSLT stylesheets must contain certain elements in the magic `xmlns:z="http://indexdata.dk/zebra/xslt/1"` namespace. The output of the XSLT indexing transformation is then parsed using DOM methods, and the contained instructions are performed on the *magic elements and their subtrees*.

For example, the output of the command

```
xsltproc xsl/oai2index.xsl one-record.xml
```

might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<z:record xmlns:z="http://indexdata.dk/zebra/xslt/1"
     z:id="oai:JTRS:CP-3290---Volume-I"
     z:rank="47896">
 <z:index name="oai_identifier" type="0">
          oai:JTRS:CP-3290---Volume-I</z:index>
 <z:index name="oai_datestamp" type="0">2004-07-09</z:index>
 <z:index name="oai_setspec" type="0">jtrs</z:index>
 <z:index name="dc_all" type="w">
    <z:index name="dc_title" type="w">Proceedings of the 4th
          International Conference and Exhibition:
          World Congress on Superconductivity - Volume I</z:index>
    <z:index name="dc_creator" type="w">Kumar Krishen and *Calvin
          Burnham, Editors</z:index>
 </z:index>
</z:record>
```

This means the following: From the original XML file `one-record.xml` (or from the XML record DOM of the same form coming from a split input file), the indexing stylesheet produces an indexing XML record, which is defined by the `record` element in the magic namespace `xmlns:z="http://indexdata.dk/zebra/xslt/1"`. Zebra uses the content of `z:id="oai:JTRS:CP-3290---Volume-I"` as internal record ID, and - in case static ranking is set - the content of `z:rank="47896"` as static rank. Following the discussion in Section 6.9 we see that this records is internally ordered lexicographically according to the value of the string `oai:JTRS:CP-3290---Volume-I47896`.

In this example, the following literal indexes are constructed:

```
oai_identifier
oai_datestamp
oai_setspec
dc_all
dc_title
dc_creator
```

where the indexing type is defined in the `type` attribute (any value from the standard configuration file `default.idx` will do). Finally, any `text()` node content recursively contained inside the `index` will be filtered through the appropriate char map for character normalization, and will be inserted in the index.

Specific to this example, we see that the single word `oai:JTRS:CP-3290---Volume-I` will be literal, byte for byte without any form of character normalization, inserted into the index named `oai:identifier`, the text `Kumar Krishen and *Calvin Burnham, Editors` will be inserted using the `w` character normalization defined in `default.idx` into the index `dc:creator` (that is, after character normalization the index will keep the individual words `kumar`, `krishen`, `and`, `calvin`, `burnham`, and `editors`), and finally both the texts `Proceedings of the 4th International Conference and Exhibition: World Congress on Superconductivity - Volume I` and `Kumar Krishen and *Calvin Burnham, Editors` will be inserted into the index `dc:all` using the same character normalization map `w`.

Finally, this example configuration can be queried using PQF queries, either transported by Z39.50, (here using a yaz-client)

```
Z> open localhost:9999
Z> elem dc
Z> form xml
Z>
Z> f @attr 1=dc_creator Kumar
Z> scan @attr 1=dc_creator adam
Z>
Z> f @attr 1=dc_title @attr 4=2 "proceeding congress  ↩
   superconductivity"
Z> scan @attr 1=dc_title abc
```

or the proprietary extensions `x-pquery` and `x-pScanClause` to SRU, and SRW

```
http://localhost:9999/?version=1.1&operation=searchRetrieve&x-pquery ↩
   =%40attr+1%3Ddc_creator+%40attr+4%3D6+%22the
```

```
http://localhost:9999/?version=1.1&operation=scan&x-pScanClause=@attr ↩
   +1=dc_date+@attr+4=2+a
```

See the section called "The SRU Server" for more information on SRU/SRW configuration, and the section called "YAZ server virtual hosts" or the YAZ CQL section for the details or the YAZ frontend server.

Notice that there are no `*.abs`, `*.est`, `*.map`, or other GRS-1 filter configuration files involves in this process, and that the literal index names are used during search and retrieval.

8.2 ALVIS Record Model Configuration

8.2.1 ALVIS Indexing Configuration

As mentioned above, there can be only one indexing stylesheet, and configuration of the indexing process is a synonym of writing an XSLT stylesheet which produces XML output containing the magic elements discussed in Section 8.1.1. Obviously, there are million of different ways to accomplish this task, and some comments and code snippets are in order to lead our Padawan's on the right track to the good side of the force.

Stylesheets can be written in the *pull* or the *push* style: *pull* means that the output XML structure is taken as starting point of the internal structure of the XSLT stylesheet, and portions of the input XML are *pulled* out and inserted into the right spots of the output XML structure. On the other side, *push* XSLT stylesheets are recursively calling their template definitions, a process which is commanded by the input XML structure, and are triggered to produce some output XML whenever some special conditions in the input stylesheets are met. The *pull* type is well-suited for input XML with strong and well-defined structure and semantics, like the following OAI indexing example, whereas the *push* type might be the only possible way to sort out deeply recursive input XML formats.

A *pull* stylesheet example used to index OAI harvested records could use some of the following template definitions:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:z="http://indexdata.dk/zebra/xslt/1"
 xmlns:oai="http://www.openarchives.org/&acro.oai;/2.0/"
 xmlns:oai_dc="http://www.openarchives.org/&acro.oai;/2.0/oai_dc/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 version="1.0">

 <xsl:output indent="yes" method="xml" version="1.0" encoding="UTF ↩
    -8"/>

  <!-- disable all default text node output -->
  <xsl:template match="text()"/>

   <!-- match on oai xml record root -->
   <xsl:template match="/">
```

```
      <z:record z:id="{normalize-space(oai:record/oai:header/oai: ↩
         identifier)}">
       <!-- you might want to use z:rank="{some &acro.xslt; function  ↩
          here}" -->
       <xsl:apply-templates/>
      </z:record>
     </xsl:template>

     <!-- &acro.oai; indexing templates -->
     <xsl:template match="oai:record/oai:header/oai:identifier">
      <z:index name="oai_identifier" type="0">
       <xsl:value-of select="."/>
      </z:index>
     </xsl:template>

     <!-- etc, etc -->

     <!-- DC specific indexing templates -->
     <xsl:template match="oai:record/oai:metadata/oai_dc:dc/dc:title">
      <z:index name="dc_title" type="w">
       <xsl:value-of select="."/>
      </z:index>
     </xsl:template>

     <!-- etc, etc -->

  </xsl:stylesheet>
```

Notice also, that the names and types of the indexes can be defined in the indexing XSLT stylesheet *dynamically according to content in the original XML records*, which has opportunities for great power and wizardry as well as grande disaster.

The following excerpt of a *push* stylesheet *might* be a good idea according to your strict control of the XML input format (due to rigorous checking against well-defined and tight RelaxNG or XML Schema's, for example):

```
  <xsl:template name="element-name-indexes">
   <z:index name="{name()}" type="w">
    <xsl:value-of select="'1'"/>
   </z:index>
  </xsl:template>
```

This template creates indexes which have the name of the working node of any input XML file, and assigns a '1' to the index. The example query `find @attr 1=xyz 1` finds all files which contain at least one `xyz` XML element. In case you can not control which element names the input files contain, you might ask for disaster and bad karma using this technique.

One variation over the theme *dynamically created indexes* will definitely be unwise:

```
<!-- match on oai xml record root -->
<xsl:template match="/">
 <z:record>

  <!-- create dynamic index name from input content -->
  <xsl:variable name="dynamic_content">
   <xsl:value-of select="oai:record/oai:header/oai:identifier"/>
  </xsl:variable>

  <!-- create zillions of indexes with unknown names -->
  <z:index name="{$dynamic_content}" type="w">
   <xsl:value-of select="oai:record/oai:metadata/oai_dc:dc"/>
  </z:index>
 </z:record>

</xsl:template>
```

Don't be tempted to cross the line to the dark side of the force, Padawan; this leads to suffering and pain, and universal disintegration of your project schedule.

8.2.2 ALVIS Exchange Formats

An exchange format can be anything which can be the outcome of an XSLT transformation, as far as the stylesheet is registered in the main Alvis XSLT filter configuration file, see Section 8.1. In principle anything that can be expressed in XML, HTML, and TEXT can be the output of a `schema` or `element set` directive during search, as long as the information comes from the *original input record XML DOM tree* (and not the transformed and *indexed* XML!!).

In addition, internal administrative information from the Zebra indexer can be accessed during record retrieval. The following example is a summary of the possibilities:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:z="http://indexdata.dk/zebra/xslt/1"
 version="1.0">

 <!-- register internal zebra parameters -->
 <xsl:param name="id" select="''"/>
 <xsl:param name="filename" select="''"/>
 <xsl:param name="score" select="''"/>
 <xsl:param name="schema" select="''"/>

 <xsl:output indent="yes" method="xml" version="1.0" encoding="UTF ↩
    -8"/>

 <!-- use then for display of internal information -->
 <xsl:template match="/">
```

```
        <z:zebra>
          <id><xsl:value-of select="$id"/></id>
          <filename><xsl:value-of select="$filename"/></filename>
          <score><xsl:value-of select="$score"/></score>
          <schema><xsl:value-of select="$schema"/></schema>
        </z:zebra>
      </xsl:template>

    </xsl:stylesheet>
```

8.2.3 ALVIS Filter OAI Indexing Example

The source code tarball contains a working Alvis filter example in the directory `examples/alvis-oai/`, which should get you started.

More example data can be harvested from any OAI compliant server, see details at the OAI http://www.openarchive web site, and the community links at http://www.openarchives.org/community/index.html. There is a tutorial found at http://www.oaforum.org/tutorial/.

Chapter 9

GRS-1 Record Model and Filter Modules

Note
The functionality of this record model has been improved and replaced by the DOM XML record model. See Chapter 7.

The record model described in this chapter applies to the fundamental, structured record type `grs`, introduced in Section 4.2.5.3.

9.1 GRS-1 Record Filters

Many basic subtypes of the *grs* type are currently available:

`grs.sgml` This is the canonical input format described Section 9.1.1. It is using simple SGML-like syntax.

`grs.marc.`*`type`* This allows Zebra to read records in the ISO2709 (MARC) encoding standard. Last parameter *`type`* names the `.abs` file (see below) which describes the specific MARC structure of the input record as well as the indexing rules.

The `grs.marc` uses an internal representation which is not XML conformant. In particular MARC tags are presented as elements with the same name. And XML elements may not start with digits. Therefore this filter is only suitable for systems returning GRS-1 and MARC records. For XML use `grs.marcxml` filter instead (see below).

The loadable `grs.marc` filter module is packaged in the GNU/Debian package `libidzebra2.-0-mod-grs-marc`

`grs.marcxml.`*`type`* This allows Zebra to read ISO2709 encoded records. Last parameter *`type`* names the `.abs` file (see below) which describes the specific MARC structure of the input record as well as the indexing rules.

The internal representation for `grs.marcxml` is the same as for MARCXML. It slightly more complicated to work with than `grs.marc` but XML conformant.

The loadable `grs.marcxml` filter module is also contained in the GNU/Debian package `libidzebra2.0-mod-grs-marc`

`grs.xml` This filter reads XML records and uses Expat to parse them and convert them into IDZebra's internal `grs` record model. Only one record per file is supported, due to the fact XML does not allow two documents to "follow" each other (there is no way to know when a document is finished). This filter is only available if Zebra is compiled with EXPAT support.

The loadable `grs.xml` filter module is packaged in the GNU/Debian package `libidzebra2.0--mod-grs-xml`

`grs.regx.filter` This enables a user-supplied Regular Expressions input filter described in Section 9.1.2.

The loadable `grs.regx` filter module is packaged in the GNU/Debian package `libidzebra2.-0-mod-grs-regx`

`grs.tcl.filter` Similar to grs.regx but using Tcl for rules, described in Section 9.1.2.

The loadable `grs.tcl` filter module is also packaged in the GNU/Debian package `libidzebra-2.0-mod-grs-regx`

9.1.1 GRS-1 Canonical Input Format

Although input data can take any form, it is sometimes useful to describe the record processing capabilities of the system in terms of a single, canonical input format that gives access to the full spectrum of structure and flexibility in the system. In Zebra, this canonical format is an "SGML-like" syntax.

To use the canonical format specify `grs.sgml` as the record type.

Consider a record describing an information resource (such a record is sometimes known as a *locator record*). It might contain a field describing the distributor of the information resource, which might in turn be partitioned into various fields providing details about the distributor, like this:

```
<Distributor>
  <Name> USGS/WRD </Name>
  <Organization> USGS/WRD </Organization>
  <Street-Address>
    U.S. GEOLOGICAL SURVEY, 505 MARQUETTE, NW
  </Street-Address>
  <City> ALBUQUERQUE </City>
  <State> NM </State>
  <Zip-Code> 87102 </Zip-Code>
  <Country> USA </Country>
  <Telephone> (505) 766-5560 </Telephone>
</Distributor>
```

The keywords surrounded by <...> are *tags*, while the sections of text in between are the *data elements*. A data element is characterized by its location in the tree that is made up by the nested elements. Each element is terminated by a closing tag - beginning with </, and containing the same symbolic tag-name as the corresponding opening tag. The general closing tag - </> - terminates the element started by the last opening tag. The structuring of elements is significant. The element *Telephone*, for instance, may be indexed and presented to the client differently, depending on whether it appears inside the *Distributor* element, or some other, structured data element such a *Supplier* element.

9.1.1.1 Record Root

The first tag in a record describes the root node of the tree that makes up the total record. In the canonical input format, the root tag should contain the name of the schema that lends context to the elements of the record (see Section 9.2). The following is a GILS record that contains only a single element (strictly speaking, that makes it an illegal GILS record, since the GILS profile includes several mandatory elements - Zebra does not validate the contents of a record against the Z39.50 profile, however - it merely attempts to match up elements of a local representation with the given schema):

```
<gils>
   <title>Zen and the Art of Motorcycle Maintenance</title>
</gils>
```

9.1.1.2 Variants

Zebra allows you to provide individual data elements in a number of *variant forms*. Examples of variant forms are textual data elements which might appear in different languages, and images which may appear in different formats or layouts. The variant system in Zebra is essentially a representation of the variant mechanism of Z39.50-1995.

The following is an example of a title element which occurs in two different languages.

```
<title>
<var lang lang "eng">
Zen and the Art of Motorcycle Maintenance</>
<var lang lang "dan">
Zen og Kunsten at Vedligeholde en Motorcykel</>
</title>
```

The syntax of the *variant element* is `<var class type value>`. The available values for the *class* and *type* fields are given by the variant set that is associated with the current schema (see Section 9.1.1.2).

Variant elements are terminated by the general end-tag </>, by the variant end-tag </var>, by the appearance of another variant tag with the same *class* and *value* settings, or by the appearance of another, normal tag. In other words, the end-tags for the variants used in the example above could have been omitted.

Variant elements can be nested. The element

```
<title>
<var lang lang "eng"><var body iana "text/plain">
Zen and the Art of Motorcycle Maintenance
</title>
```

Associates two variant components to the variant list for the title element.

Given the nesting rules described above, we could write

```
<title>
<var body iana "text/plain>
<var lang lang "eng">
Zen and the Art of Motorcycle Maintenance
<var lang lang "dan">
```

```
        Zen og Kunsten at Vedligeholde en Motorcykel
        </title>
```

The title element above comes in two variants. Both have the IANA body type "text/plain", but one is in English, and the other in Danish. The client, using the element selection mechanism of Z39.50, can retrieve information about the available variant forms of data elements, or it can select specific variants based on the requirements of the end-user.

9.1.2 GRS-1 REGX And TCL Input Filters

In order to handle general input formats, Zebra allows the operator to define filters which read individual records in their native format and produce an internal representation that the system can work with.

Input filters are ASCII files, generally with the suffix `.flt`. The system looks for the files in the directories given in the *profilePath* setting in the `zebra.cfg` files. The record type for the filter is `grs.regx.`*filter-filename* (fundamental type `grs`, file read type `regx`, argument *filter-filename*).

Generally, an input filter consists of a sequence of rules, where each rule consists of a sequence of expressions, followed by an action. The expressions are evaluated against the contents of the input record, and the actions normally contribute to the generation of an internal representation of the record.

An expression can be either of the following:

`INIT` The action associated with this expression is evaluated exactly once in the lifetime of the application, before any records are read. It can be used in conjunction with an action that initializes tables or other resources that are used in the processing of input records.

`BEGIN` Matches the beginning of the record. It can be used to initialize variables, etc. Typically, the *BEGIN* rule is also used to establish the root node of the record.

`END` Matches the end of the record - when all of the contents of the record has been processed.

`/reg/` Matches regular expression pattern *`reg`* from the input record. The operators supported are the same as for regular expression queries. Refer to Section 5.3.6.

`BODY` This keyword may only be used between two patterns. It matches everything between (not including) those patterns.

`FINISH` The expression associated with this pattern is evaluated once, before the application terminates. It can be used to release system resources - typically ones allocated in the *INIT* step.

An action is surrounded by curly braces ({...}), and consists of a sequence of statements. Statements may be separated by newlines or semicolons (;). Within actions, the strings that matched the expressions immediately preceding the action can be referred to as $0, $1, $2, etc.

The available statements are:

begin ***`type [parameter ... ]`*** Begin a new data element. The *`type`* is one of the following:

record Begin a new record. The following parameter should be the name of the schema that describes the structure of the record, e.g., `gils` or `wais` (see below). The `begin record` call should precede any other use of the *`begin`* statement.

element Begin a new tagged element. The parameter is the name of the tag. If the tag is not matched anywhere in the tagsets referenced by the current schema, it is treated as a local string tag.

variant Begin a new node in a variant tree. The parameters are *`class type value`*.

data *`parameter`* Create a data element. The concatenated arguments make up the value of the data element. The option `-text` signals that the layout (whitespace) of the data should be retained for transmission. The option `-element` *`tag`* wraps the data up in the *`tag`*. The use of the `-element` option is equivalent to preceding the command with a *`begin element`* command, and following it with the *`end`* command.

end *`[type]`* Close a tagged element. If no parameter is given, the last element on the stack is terminated. The first parameter, if any, is a type name, similar to the *`begin`* statement. For the *`element`* type, a tag name can be provided to terminate a specific tag.

unread *`no`* Move the input pointer to the offset of first character that match rule given by *`no`*. The first rule from left-to-right is numbered zero, the second rule is named 1 and so on.

The following input filter reads a Usenet news file, producing a record in the WAIS schema. Note that the body of a news posting is separated from the list of headers by a blank line (or rather a sequence of two newline characters.

```
BEGIN                { begin record wais }

/^From:/ BODY /$/    { data -element name $1 }
/^Subject:/ BODY /$/ { data -element title $1 }
/^Date:/ BODY /$/    { data -element lastModified $1 }
/\n\n/ BODY END      {
   begin element bodyOfDisplay
   begin variant body iana "text/plain"
   data -text $1
   end record
}
```

If Zebra is compiled with support for Tcl enabled, the statements described above are supplemented with a complete scripting environment, including control structures (conditional expressions and loop constructs), and powerful string manipulation mechanisms for modifying the elements of a record.

9.2 GRS-1 Internal Record Representation

When records are manipulated by the system, they're represented in a tree-structure, with data elements at the leaf nodes, and tags or variant components at the non-leaf nodes. The root-node identifies the schema that lends context to the tagging and structuring of the record. Imagine a simple record, consisting of a 'title' element and an 'author' element:

```
ROOT
   TITLE     "Zen and the Art of Motorcycle Maintenance"
   AUTHOR    "Robert Pirsig"
```

A slightly more complex record would have the author element consist of two elements, a surname and a first name:

```
ROOT
   TITLE  "Zen and the Art of Motorcycle Maintenance"
   AUTHOR
      FIRST-NAME "Robert"
      SURNAME    "Pirsig"
```

The root of the record will refer to the record schema that describes the structuring of this particular record. The schema defines the element tags (TITLE, FIRST-NAME, etc.) that may occur in the record, as well as the structuring (SURNAME should appear below AUTHOR, etc.). In addition, the schema establishes element set names that are used by the client to request a subset of the elements of a given record. The schema may also establish rules for converting the record to a different schema, by stating, for each element, a mapping to a different tag path.

9.2.1 Tagged Elements

A data element is characterized by its tag, and its position in the structure of the record. For instance, while the tag "telephone number" may be used different places in a record, we may need to distinguish between these occurrences, both for searching and presentation purposes. For instance, while the phone numbers for the "customer" and the "service provider" are both representatives for the same type of resource (a telephone number), it is essential that they be kept separate. The record schema provides the structure of the record, and names each data element (defined by the sequence of tags - the tag path - by which the element can be reached from the root of the record).

9.2.2 Variants

The children of a tag node may be either more tag nodes, a data node (possibly accompanied by tag nodes), or a tree of variant nodes. The children of variant nodes are either more variant nodes or a data node (possibly accompanied by more variant nodes). Each leaf node, which is normally a data node, corresponds to a *variant form* of the tagged element identified by the tag which parents the variant tree. The following title element occurs in two different languages:

```
 VARIANT LANG=ENG  "War and Peace"
 TITLE
 VARIANT LANG=DAN  "Krig og Fred"
```

Which of the two elements are transmitted to the client by the server depends on the specifications provided by the client, if any.

In practice, each variant node is associated with a triple of class, type, value, corresponding to the variant mechanism of Z39.50.

9.2.3 Data Elements

Data nodes have no children (they are always leaf nodes in the record tree).

9.3 GRS-1 Record Model Configuration

The following sections describe the configuration files that govern the internal management of `grs` records. The system searches for the files in the directories specified by the *profilePath* setting in the `zebra.cfg` file.

9.3.1 The Abstract Syntax

The abstract syntax definition (also known as an Abstract Record Structure, or ARS) is the focal point of the record schema description. For a given schema, the ABS file may state any or all of the following:

- The object identifier of the Z39.50 schema associated with the ARS, so that it can be referred to by the client.
- The attribute set (which can possibly be a compound of multiple sets) which applies in the profile. This is used when indexing and searching the records belonging to the given profile.
- The tag set (again, this can consist of several different sets). This is used when reading the records from a file, to recognize the different tags, and when transmitting the record to the client - mapping the tags to their numerical representation, if they are known.
- The variant set which is used in the profile. This provides a vocabulary for specifying the *forms* of data that appear inside the records.
- Element set names, which are a shorthand way for the client to ask for a subset of the data elements contained in a record. Element set names, in the retrieval module, are mapped to *element specifications*, which contain information equivalent to the *Espec-1* syntax of Z39.50.
- Map tables, which may specify mappings to *other* database profiles, if desired.
- Possibly, a set of rules describing the mapping of elements to a MARC representation.
- A list of element descriptions (this is the actual ARS of the schema, in Z39.50 terms), which lists the ways in which the various tags can be used and organized hierarchically.

Several of the entries above simply refer to other files, which describe the given objects.

9.3.2 The Configuration Files

This section describes the syntax and use of the various tables which are used by the retrieval module.

The number of different file types may appear daunting at first, but each type corresponds fairly clearly to a single aspect of the Z39.50 retrieval facilities. Further, the average database administrator, who is simply reusing an existing profile for which tables already exist, shouldn't have to worry too much about the contents of these tables.

Generally, the files are simple ASCII files, which can be maintained using any text editor. Blank lines, and lines beginning with a (#) are ignored. Any characters on a line followed by a (#) are also ignored. All other lines contain *directives*, which provide some setting or value to the system. Generally, settings are characterized by a single keyword, identifying the setting, followed by a number of parameters. Some settings are repeatable (r), while others may occur only once in a file. Some settings are optional (o), while others again are mandatory (m).

9.3.3 The Abstract Syntax (.abs) Files

The name of this file type is slightly misleading in Z39.50 terms, since, apart from the actual abstract syntax of the profile, it also includes most of the other definitions that go into a database profile.

When a record in the canonical, SGML-like format is read from a file or from the database, the first tag of the file should reference the profile that governs the layout of the record. If the first tag of the record is, say, `<gils>`, the system will look for the profile definition in the file `gils.abs`. Profile definitions are cached, so they only have to be read once during the lifespan of the current process.

When writing your own input filters, the *record-begin* command introduces the profile, and should always be called first thing when introducing a new record.

The file may contain the following directives:

name ***symbolic-name*** (m) This provides a shorthand name or description for the profile. Mostly useful for diagnostic purposes.

reference ***OID-name*** (m) The reference name of the OID for the profile. The reference names can be found in the *util* module of YAZ.

attset ***filename*** (m) The attribute set that is used for indexing and searching records belonging to this profile.

tagset ***filename*** (o) The tag set (if any) that describe that fields of the records.

varset ***filename*** (o) The variant set used in the profile.

maptab ***filename*** (o,r) This points to a conversion table that might be used if the client asks for the record in a different schema from the native one.

marc ***filename*** (o) Points to a file containing parameters for representing the record contents in the ISO2709 syntax. Read the description of the MARC representation facility below.

esetname ***name filename*** (o,r) Associates the given element set name with an element selection file. If an (@) is given in place of the filename, this corresponds to a null mapping for the given element set name.

all ***tags*** (o) This directive specifies a list of attributes which should be appended to the attribute list given for each element. The effect is to make every single element in the abstract syntax searchable by way of the given attributes. This directive provides an efficient way of supporting free-text searching across all elements. However, it does increase the size of the index significantly. The attributes can be qualified with a structure, as in the *elm* directive below.

elm ***path name attributes*** (o,r) Adds an element to the abstract record syntax of the schema. The *path* follows the syntax which is suggested by the Z39.50 document - that is, a sequence of tags separated by slashes (/). Each tag is given as a comma-separated pair of tag type and -value surrounded by parenthesis. The *name* is the name of the element, and the *attributes* specifies which attributes to use when indexing the element in a comma-separated list. A ! in place of the attribute name is equivalent to specifying an attribute name identical to the element name. A − in place of the attribute name specifies that no indexing is to take place for the given element. The attributes can be qualified with *field types* to specify which character set should govern the indexing procedure for that field. The same data element may be indexed into several different fields, using different character set definitions. See the Chapter 10. The default field type is `w` for *word*.

xelm ***xpath attributes*** Specifies indexing for record nodes given by *xpath*. Unlike directive elm, this directive allows you to index attribute contents. The *xpath* uses a syntax similar to XPath. The *attributes* have same syntax and meaning as directive elm, except that operator ! refers to the nodes selected by *xpath*.

melm ***field$subfield attributes*** This directive is specifically for MARC-formatted records, ingested either in the form of MARCXML documents, or in the ISO2709/Z39.2 format using the grs.marcxml input filter. You can specify indexing rules for any subfield, or you can leave off the *$subfield* part and specify default rules for all subfields of the given field (note: default rules should come after any subfield-specific rules in the configuration file). The *attributes* have the same syntax and meaning as for the 'elm' directive above.

encoding ***encodingname*** This directive specifies character encoding for external records. For records such as XML that specifies encoding within the file via a header this directive is ignored. If neither this directive is given, nor an encoding is set within external records, ISO-8859-1 encoding is assumed.

xpath **`enable/disable`** If this directive is followed by `enable`, then extra indexing is performed to allow for XPath-like queries. If this directive is not specified - equivalent to `disable` - no extra XPath-indexing is performed.

systag ***systemTag actualTag*** Specifies what information, if any, Zebra should automatically include in retrieval records for the ``system fields'' that it supports. *systemTag* may be any of the following:

`rank` An integer indicating the relevance-ranking score assigned to the record.

`sysno` An automatically generated identifier for the record, unique within this database. It is represented by the `<localControlNumber>` element in XML and the `(1,14)` tag in GRS-1.

`size` The size, in bytes, of the retrieved record.

The *actualTag* parameter may be `none` to indicate that the named element should be omitted from retrieval records.

Note
The mechanism for controlling indexing is not adequate for complex databases, and will probably be moved into a separate configuration table eventually.

The following is an excerpt from the abstract syntax file for the GILS profile.

```
name gils
reference GILS-schema
attset gils.att
tagset gils.tag
varset var1.var

maptab gils-usmarc.map

# Element set names

esetname VARIANT gils-variant.est  # for WAIS-compliance
esetname B gils-b.est
esetname G gils-g.est
esetname F @

elm (1,10)               rank                       -
elm (1,12)               url                        -
elm (1,14)               localControlNumber     Local-number
elm (1,16)               dateOfLastModification Date/time-last- ↩
   modified
elm (2,1)                title                      w:!,p:!
elm (4,1)                controlIdentifier      Identifier-standard
elm (2,6)                abstract               Abstract
elm (4,51)               purpose                    !
elm (4,52)               originator                 -
elm (4,53)               accessConstraints          !
elm (4,54)               useConstraints             !
elm (4,70)               availability               -
elm (4,70)/(4,90)        distributor                -
elm (4,70)/(4,90)/(2,7)  distributorName            !
elm (4,70)/(4,90)/(2,10) distributorOrganization    !
elm (4,70)/(4,90)/(4,2)  distributorStreetAddress   !
elm (4,70)/(4,90)/(4,3)  distributorCity            !
```

9.3.4 The Attribute Set (.att) Files

This file type describes the *Use* elements of an attribute set. It contains the following directives.

name ***symbolic-name*** (m) This provides a shorthand name or description for the attribute set. Mostly useful for diagnostic purposes.

reference ***OID-name*** (m) The reference name of the OID for the attribute set. The reference names can be found in the *util* module of *YAZ*.

include ***filename*** (o,r) This directive is used to include another attribute set as a part of the current one. This is used when a new attribute set is defined as an extension to another set. For instance, many new attribute sets are defined as extensions to the *bib-1* set. This is an important feature of the retrieval system of Z39.50, as it ensures the highest possible level of interoperability, as those access points of your database which are derived from the external set (say, bib-1) can be used even by clients who are unaware of the new set.

att ***att-value att-name [local-value]*** (o,r) This repeatable directive introduces a new attribute to the set. The attribute value is stored in the index (unless a *local-value* is given, in which case this is stored). The name is used to refer to the attribute from the *abstract syntax*.

This is an excerpt from the GILS attribute set definition. Notice how the file describing the *bib-1* attribute set is referenced.

```
name gils
reference GILS-attset
include bib1.att

att 2001    distributorName
att 2002    indextermsControlled
att 2003    purpose
att 2004    accessConstraints
att 2005    useConstraints
```

9.3.5 The Tag Set (.tag) Files

This file type defines the tagset of the profile, possibly by referencing other tag sets (most tag sets, for instance, will include tagsetG and tagsetM from the Z39.50 specification. The file may contain the following directives.

name ***symbolic-name*** (m) This provides a shorthand name or description for the tag set. Mostly useful for diagnostic purposes.

reference ***OID-name*** (o) The reference name of the OID for the tag set. The reference names can be found in the *util* module of *YAZ*. The directive is optional, since not all tag sets are registered outside of their schema.

type ***integer*** (m) The type number of the tagset within the schema profile (note: this specification really should belong to the .abs file. This will be fixed in a future release).

include ***filename*** (o,r) This directive is used to include the definitions of other tag sets into the current one.

tag ***number names type*** (o,r) Introduces a new tag to the set. The *number* is the tag number as used in the protocol (there is currently no mechanism for specifying string tags at this point, but this would be quick work to add). The *names* parameter is a list of names by which the tag should be recognized in the input file format. The names should be separated by slashes (/). The *type* is the recommended data type of the tag. It should be one of the following:

- structured
- string
- numeric
- bool
- oid
- generalizedtime
- intunit
- int
- octetstring
- null

The following is an excerpt from the TagsetG definition file.

```
name tagsetg
reference TagsetG
type 2

tag 1 title   string
tag 2 author    string
tag 3 publicationPlace string
tag 4 publicationDate string
tag 5 documentId  string
tag 6 abstract  string
tag 7 name    string
tag 8 date    generalizedtime
tag 9 bodyOfDisplay string
tag 10  organization  string
```

9.3.6 The Variant Set (.var) Files

The variant set file is a straightforward representation of the variant set definitions associated with the protocol. At present, only the *Variant-1* set is known.

These are the directives allowed in the file.

name ***symbolic-name*** (m) This provides a shorthand name or description for the variant set. Mostly useful for diagnostic purposes.

reference ***OID-name*** (o) The reference name of the OID for the variant set, if one is required. The reference names can be found in the *util* module of *YAZ*.

class ***integer class-name*** (m,r) Introduces a new class to the variant set.

type ***integer type-name datatype*** (m,r) Addes a new type to the current class (the one introduced by the most recent *class* directive). The type names belong to the same name space as the one used in the tag set definition file.

The following is an excerpt from the file describing the variant set *Variant-1*.

```
name variant-1
reference Variant-1

class 1 variantId

type  1 variantId   octetstring

class 2 body

type  1 iana      string
type  2 z39.50      string
type  3 other     string
```

9.3.7 The Element Set (.est) Files

The element set specification files describe a selection of a subset of the elements of a database record. The element selection mechanism is equivalent to the one supplied by the *Espec-1* syntax of the Z39.50 specification. In fact, the internal representation of an element set specification is identical to the *Espec-1* structure, and we'll refer you to the description of that structure for most of the detailed semantics of the directives below.

Note
Not all of the Espec-1 functionality has been implemented yet. The fields that are mentioned below all work as expected, unless otherwise is noted.

The directives available in the element set file are as follows:

defaultVariantSetId ***OID-name*** (o) If variants are used in the following, this should provide the name of the variantset used (it's not currently possible to specify a different set in the individual variant request). In almost all cases (certainly all profiles known to us), the name `Variant-1` should be given here.

defaultVariantRequest ***variant-request*** (o) This directive provides a default variant request for use when the individual element requests (see below) do not contain a variant request. Variant requests consist of a blank-separated list of variant components. A variant component is a comma-separated, parenthesized triple of variant class, type, and value (the two former values being represented as integers). The value can currently only be entered as a string (this will change to depend on the definition of the variant in question). The special value (@) is interpreted as a null value, however.

simpleElement ***path ['variant' variant-request]*** (o,r) This corresponds to a simple element request in *Espec-1*. The path consists of a sequence of tag-selectors, where each of these can consist of either:

- A simple tag, consisting of a comma-separated type-value pair in parenthesis, possibly followed by a colon (:) followed by an occurrences-specification (see below). The tag-value can be a number or a string. If the first character is an apostrophe ('), this forces the value to be interpreted as a string, even if it appears to be numerical.
- A WildThing, represented as a question mark (?), possibly followed by a colon (:) followed by an occurrences specification (see below).
- A WildPath, represented as an asterisk (*). Note that the last element of the path should not be a wildPath (wildpaths don't work in this version).

The occurrences-specification can be either the string `all`, the string `last`, or an explicit value-range. The value-range is represented as an integer (the starting point), possibly followed by a plus (+) and a second integer (the number of elements, default being one).

The variant-request has the same syntax as the defaultVariantRequest above. Note that it may sometimes be useful to give an empty variant request, simply to disable the default for a specific set of fields (we aren't certain if this is proper *Espec-1*, but it works in this implementation).

The following is an example of an element specification belonging to the GILS profile.

```
simpleelement (1,10)
simpleelement (1,12)
simpleelement (2,1)
simpleelement (1,14)
simpleelement (4,1)
simpleelement (4,52)
```

9.3.8 The Schema Mapping (.map) Files

Sometimes, the client might want to receive a database record in a schema that differs from the native schema of the record. For instance, a client might only know how to process WAIS records, while the database record is represented in a more specific schema, such as GILS. In this module, a mapping of data to one of the MARC formats is also thought of as a schema mapping (mapping the elements of the record into fields consistent with the given MARC specification, prior to actually converting the data to the ISO2709). This use of the object identifier for USMARC as a schema identifier represents an overloading of the OID which might not be entirely proper. However, it represents the dual role of schema and record syntax which is assumed by the MARC family in Z39.50.

These are the directives of the schema mapping file format:

targetName ***name*** (m) A symbolic name for the target schema of the table. Useful mostly for diagnostic purposes.

targetRef ***OID-name*** (m) An OID name for the target schema. This is used, for instance, by a server receiving a request to present a record in a different schema from the native one. The name, again, is found in the *oid* module of *YAZ*.

map ***element-name target-path*** (o,r) Adds an element mapping rule to the table.

9.3.9 The MARC (ISO2709) Representation (.mar) Files

This file provides rules for representing a record in the ISO2709 format. The rules pertain mostly to the values of the constant-length header of the record.

9.4 GRS-1 Exchange Formats

Converting records from the internal structure to an exchange format is largely an automatic process. Currently, the following exchange formats are supported:

- GRS-1. The internal representation is based on GRS-1/XML, so the conversion here is straightforward. The system will create applied variant and supported variant lists as required, if a record contains variant information.
- XML. The internal representation is based on GRS-1/XML so the mapping is trivial. Note that XML schemas, preprocessing instructions and comments are not part of the internal representation and therefore will never be part of a generated XML record. Future versions of the Zebra will support that.
- SUTRS. Again, the mapping is fairly straightforward. Indentation is used to show the hierarchical structure of the record. All "GRS-1" type records support both the GRS-1 and SUTRS representations.
- ISO2709-based formats (USMARC, etc.). Only records with a two-level structure (corresponding to fields and subfields) can be directly mapped to ISO2709. For records with a different structuring (e.g., GILS), the representation in a structure like USMARC involves a schema-mapping (see Section 9.3.8), to an "implied" USMARC schema (implied, because there is no formal schema which specifies the use of the USMARC fields outside of ISO2709). The resultant, two-level record is then mapped directly from the internal representation to ISO2709. See the GILS schema definition files for a detailed example of this approach.
- Explain. This representation is only available for records belonging to the Explain schema.
- Summary. This ASN-1 based structure is only available for records belonging to the Summary schema - or schema which provide a mapping to this schema (see the description of the schema mapping facility above).
- SOIF. Support for this syntax is experimental, and is currently keyed to a private Index Data OID (1.2.840.10003.5.1000.81.2). All abstract syntaxes can be mapped to the SOIF format, although nested elements are represented by concatenation of the tag names at each level.

9.5 Extended indexing of MARC records

Extended indexing of MARC records will help you if you need index a combination of subfields, or index only a part of the whole field, or use during indexing process embedded fields of MARC record.

Extended indexing of MARC records additionally allows:

- to index data in LEADER of MARC record

- to index data in control fields (with fixed length)
- to use during indexing the values of indicators
- to index linked fields for UNIMARC based formats

Note
In compare with simple indexing process the extended indexing may increase (about 2-3 times) the time of indexing process for MARC records.

9.5.1 The index-formula

At the beginning, we have to define the term *index-formula* for MARC records. This term helps to understand the notation of extended indexing of MARC records by Zebra. Our definition is based on the document "The table of conformity for Z39.50 use attributes and RUSMARC fields". The document is available only in Russian language.

The *index-formula* is the combination of subfields presented in such way:

```
71-00$a, $g, $h ($c){.$b ($c)} , (1)
```

We know that Zebra supports a BIB-1 attribute - right truncation. In this case, the *index-formula* (1) consists from forms, defined in the same way as (1)

```
71-00$a, $g, $h
71-00$a, $g
71-00$a
```

Note
The original MARC record may be without some elements, which included in *index-formula*.

This notation includes such operands as:

It means whitespace character.

- The position may contain any value, defined by MARC format. For example, *index-formula*

```
70-#1$a, $g , (2)
```

includes

```
700#1$a, $g
701#1$a, $g
702#1$a, $g
```

{...} The repeatable elements are defined in figure-brackets {}. For example, *index-formula*

```
71-00$a, $g, $h ($c){.$b ($c)} , (3)
```

includes

```
71-00$a, $g, $h ($c). $b ($c)
71-00$a, $g, $h ($c). $b ($c). $b ($c)
71-00$a, $g, $h ($c). $b ($c). $b ($c). $b ($c)
```

Note
All another operands are the same as accepted in MARC world.

9.5.2 Notation of *index-formula* for Zebra

Extended indexing overloads `path` of `elm` definition in abstract syntax file of Zebra (`.abs` file). It means that names beginning with `"mc-"` are interpreted by Zebra as *index-formula*. The database index is created and linked with *access point* (BIB-1 use attribute) according to this formula.

For example, *index-formula*

```
71-00$a, $g, $h ($c){.$b ($c)} , (4)
```

in `.abs` file looks like:

```
mc-71.00_$a,_$g,_$h_(_$c_){.$b_(_$c_)}
```

The notation of *index-formula* uses the operands:

_ It means whitespace character.

. The position may contain any value, defined by MARC format. For example, *index-formula*

```
70-#1$a, $g , (5)
```

matches `mc-70._1_$a,_$g_` and includes

```
700_1_$a,_$g_
701_1_$a,_$g_
702_1_$a,_$g_
```

{...} The repeatable elements are defined in figure-brackets {}. For example, *index-formula*

```
71#00$a, $g, $h ($c) {.$b ($c)} , (6)
```

matches `mc-71.00_$a,_$g,_$h_(_$c_){.$b_(_$c_)}` and includes

```
71.00_$a,_$g,_$h_(_$c_).$b_(_$c_)
71.00_$a,_$g,_$h_(_$c_).$b_(_$c_).$b_(_$c_)
71.00_$a,_$g,_$h_(_$c_).$b_(_$c_).$b_(_$c_).$b_(_$c_)
```

<...> Embedded *index-formula* (for linked fields) is between <>. For example, *index-formula*

```
4--#-$170-#1$a, $g ($c) , (7)
```

matches `mc-4..._.._$1<70._1_$a,_$g_(_$c_)>_` and includes

```
463_._$1<70._1_$a,_$g_(_$c_)>_
```

Note
All another operands are the same as accepted in MARC world.

9.5.2.1 Examples

1. indexing LEADER

   You need to use keyword "ldr" to index leader. For example, indexing data from 6th and 7th position of LEADER

   ```
   elm mc-ldr[6] Record-type !
   elm mc-ldr[7] Bib-level   !
   ```

2. indexing data from control fields

   indexing date (the time added to database)

   ```
   elm mc-008[0-5] Date/time-added-to-db !
   ```

   or for RUSMARC (this data included in 100th field)

   ```
   elm mc-100___$a[0-7]_ Date/time-added-to-db !
   ```

3. using indicators while indexing

   For RUSMARC *index-formula* `70-#1$a, $g` matches

   ```
   elm 70._1_$a,_$g_ Author !:w,!:p
   ```

   When Zebra finds a field according to `"70."` pattern it checks the indicators. In this case the value of first indicator doesn't mater, but the value of second one must be whitespace, in another case a field is not indexed.

4. indexing embedded (linked) fields for UNIMARC based formats

   For RUSMARC *index-formula* `4--#-$170-#1$a, $g ($c)` matches

   ```
   elm mc-4..._.._$1<70._1_$a,_$g_(_$c_)>_ Author !:w,!:p
   ```

   Data are extracted from record if the field matches to `"4..._."` pattern and data in linked field match to embedded *index-formula* `70._1_$a,_$g_(_$c_)`.

Chapter 10

Field Structure and Character Sets

In order to provide a flexible approach to national character set handling, Zebra allows the administrator to configure the set up the system to handle any 8-bit character set — including sets that require multi-octet diacritics or other multi-octet characters. The definition of a character set includes a specification of the permissible values, their sort order (this affects the display in the SCAN function), and relationships between upper- and lowercase characters. Finally, the definition includes the specification of space characters for the set.

The operator can define different character sets for different fields, typical examples being standard text fields, numerical fields, and special-purpose fields such as WWW-style linkages (URx).

Zebra 1.3 and Zebra versions 2.0.18 and earlier required that the field type is a single character, e.g. `w` (for word), and `p` for phrase. Zebra 2.0.20 and later allow field types to be any string. This allows for greater flexibility - in particular per-locale (language) fields can be defined.

Version 2.1 of Zebra can also be configured - per field - to use the ICU library to perform tokenization and normalization of strings. This is an alternative to the "charmap" files which has been part of Zebra since its first release.

10.1 The default.idx file

The field types, and hence character sets, are associated with data elements by the indexing rules (say `title:w`) in the various filters. Fields are defined in a field definition file which, by default, is called `default.idx`. This file provides the association between field type codes and the character map files (with the .chr suffix). The format of the .idx file is as follows

index ***field type code*** This directive introduces a new search index code. The argument is a one-character code to be used in the .abs files to select this particular index type. An index, roughly, corresponds to a particular structure attribute during search. Refer to the section called "Z39.50 Search".

sort ***field code type*** This directive introduces a sort index. The argument is a one-character code to be used in the .abs fie to select this particular index type. The corresponding use attribute must be used in the sort request to refer to this particular sort index. The corresponding character map (see below) is used in the sort process.

completeness *boolean* This directive enables or disables complete field indexing. The value of the *boolean* should be 0 (disable) or 1. If completeness is enabled, the index entry will contain the complete contents of the field (up to a limit), with words (non-space characters) separated by single space characters (normalized to " " on display). When completeness is disabled, each word is indexed as a separate entry. Complete subfield indexing is most useful for fields which are typically browsed (e.g., titles, authors, or subjects), or instances where a match on a complete subfield is essential (e.g., exact title searching). For fields where completeness is disabled, the search engine will interpret a search containing space characters as a word proximity search.

firstinfield *boolean* This directive enables or disables first-in-field indexing. The value of the *boolean* should be 0 (disable) or 1.

alwaysmatches *boolean* This directive enables or disables alwaysmatches indexing. The value of the *boolean* should be 0 (disable) or 1.

charmap *filename* This is the filename of the character map to be used for this index for field type. See Section 10.2 for details.

icuchain *filename* Specifies the filename with ICU tokenization and normalization rules. See Section 10.3 for details. Using icuchain for a field type is an alternative to charmap. It does not make sense to define both icuchain and charmap for the same field type.

Example 10.1 Field types

Following are three excerpts of the standard `tab/default.idx` configuration file. Notice that the `index` and `sort` are grouping directives, which bind all other following directives to them:

```
# Traditional word index
# Used if completeness is 'incomplete field' (@attr 6=1) and
# structure is word/phrase/word-list/free-form-text/document-text
index w
completeness 0
position 1
alwaysmatches 1
firstinfield 1
charmap string.chr

...

# Null map index (no mapping at all)
# Used if structure=key (@attr 4=3)
index 0
completeness 0
position 1
charmap @

...

# Sort register
sort s
completeness 1
charmap string.chr
```

10.2 Charmap Files

The character map files are used to define the word tokenization and character normalization performed before inserting text into the inverse indexes. Zebra ships with the predefined character map files `tab/*.chr`. Users are allowed to add and/or modify maps according to their needs.

The contents of the character map files are structured as follows:

encoding *`encoding-name`* This directive must be at the very beginning of the file, and it specifies the character encoding used in the entire file. If omitted, the encoding `ISO-8859-1` is assumed.

For example, one of the test files found at `test/rusmarc/tab/string.chr` contains the following encoding directive:

```
encoding koi8-r
```

and the test file `test/charmap/string.utf8.chr` is encoded in UTF-8:

```
encoding utf-8
```

File name	Intended type	Description
`numeric.chr`	`:n`	Numeric digit tokenization and normalization map. All characters not in the set `-{0-9}.,` will be suppressed. Note that floating point numbers are processed fine, but scientific exponential numbers are trashed.
`scan.chr`	`:w or :p`	Word tokenization char map for Scandinavian languages. This one resembles the generic word tokenization character map `tab/string.chr`, the main differences are sorting of the special characters `üzæäøöå` and equivalence maps according to Scandinavian language rules.
`string.chr`	`:w or :p`	General word tokenization and normalization character map, mostly useful for English texts. Use this to derive your own language tokenization and normalization derivatives.
`urx.chr`	`:u`	URL parsing and tokenization character map.
`@`	`:0`	Do-nothing character map used for literal binary indexing. There is no existing file associated to it, and there is no normalization or tokenization performed at all.

Table 10.1: Character maps predefined in Zebra

lowercase *value-set* This directive introduces the basic value set of the field type. The format is an ordered list (without spaces) of the characters which may occur in "words" of the given type. The order of the entries in the list determines the sort order of the index. In addition to single characters, the following combinations are legal:

- Backslashes may be used to introduce three-digit octal, or two-digit hex representations of single characters (preceded by `x`). In addition, the combinations \\, \\r, \\n, \\t, \\s (space — remember that real space-characters may not occur in the value definition), and \\ are recognized, with their usual interpretation.
- Curly braces {} may be used to enclose ranges of single characters (possibly using the escape convention described in the preceding point), e.g., {a-z} to introduce the standard range of ASCII characters. Note that the interpretation of such a range depends on the concrete representation in your local, physical character set.
- parentheses () may be used to enclose multi-byte characters - e.g., diacritics or special national combinations (e.g., Spanish "ll"). When found in the input stream (or a search term), these characters are viewed and sorted as a single character, with a sorting value depending on the position of the group in the value statement.

For example, `scan.chr` contains the following lowercase normalization and sorting order:

```
lowercase {0-9}{a-y}üzæäøöå
```

uppercase *value-set* This directive introduces the upper-case equivalences to the value set (if any). The number and order of the entries in the list should be the same as in the `lowercase` directive.

For example, `scan.chr` contains the following uppercase equivalent:

```
uppercase {0-9}{A-Y}ÜZÆÄØÖÅ
```

space *value-set* This directive introduces the character which separate words in the input stream. Depending on the completeness mode of the field in question, these characters either terminate an index entry, or delimit individual "words" in the input stream. The order of the elements is not significant — otherwise the representation is the same as for the `uppercase` and `lowercase` directives.

For example, `scan.chr` contains the following space instruction:

```
space {\001-\040}!"#$%&'\()*+,-./:;<=>?@\[\\]^_`\{|}~
```

map *value-set target* This directive introduces a mapping between each of the members of the value-set on the left to the character on the right. The character on the right must occur in the value set (the `lowercase` directive) of the character set, but it may be a parenthesis-enclosed multi-octet character. This directive may be used to map diacritics to their base characters, or to map HTML-style character-representations to their natural form, etc. The map directive can also be used to ignore leading articles in searching and/or sorting, and to perform other special transformations.

For example, `scan.chr` contains the following map instructions among others, to make sure that HTML entity encoded Danish special characters are mapped to the equivalent Latin-1 characters:

```
map (&aelig;)      æ
map (&oslash;)     ø
map (&aring;)      å
```

In addition to specifying sort orders, space (blank) handling, and upper/lowercase folding, you can also use the character map files to make Zebra ignore leading articles in sorting records, or when doing complete field searching.

This is done using the `map` directive in the character map file. In a nutshell, what you do is map certain sequences of characters, when they occur *in the beginning of a field*, to a space. Assuming that the character "@" is defined as a space character in your file, you can do:

```
map (^The\s) @
map (^the\s) @
```

The effect of these directives is to map either 'the' or 'The', followed by a space character, to a space. The hat ^ character denotes beginning-of-field only when complete-subfield indexing or sort indexing is taking place; otherwise, it is treated just as any other character.

Because the `default.idx` file can be used to associate different character maps with different indexing types -- and you can create additional indexing types, should the need arise -- it is possible to specify that leading articles should be ignored either in sorting, in complete-field searching, or both.

If you ignore certain prefixes in sorting, then these will be eliminated from the index, and sorting will take place as if they weren't there. However, if you set the system up to ignore certain prefixes in *searching*, then these are deleted both from the indexes and from query terms, when the client specifies complete-field searching. This has the effect that a search for 'the science journal' and 'science journal' would both produce the same results.

equivalent *`value-set`* This directive introduces equivalence classes of characters and/or strings for sorting purposes only. It resembles the map directive, but does not affect search and retrieval indexing, but only sorting order under present requests.

For example, `scan.chr` contains the following equivalent sorting instructions, which can be uncommented:

```
# equivalent æä(ae)
# equivalent øö(oe)
# equivalent å(aa)
# equivalent uü
```

10.3 ICU Chain Files

The ICU chain files defines a *chain* of rules which specify the conversion process to be carried out for each record string for indexing.

Both searching and sorting is based on the *sort* normalization that ICU provides. This means that scan and sort will return terms in the sort order given by ICU.

Zebra is using YAZ' ICU wrapper. Refer to the yaz-icu man page for documentation about the ICU chain rules.

Tip
Use the yaz-icu program to test your icuchain rules.

Example 10.2 Indexing Greek text

Consider a system where all "regular" text is to be indexed using as Greek (locale: EL). We would have to change our index type file - to read

```
# Index greek words
index w
completeness 0
position 1
alwaysmatches 1
firstinfield 1
icuahain greek.xml
..
```

The ICU chain file `greek.xml` could look as follows:

```
<icu_chain locale="el">
<transform rule="[:Control:] Any-Remove"/>
<tokenize rule="l"/>
<transform rule="[[:WhiteSpace:][:Punctuation:]] Remove"/>
<display/>
<casemap rule="l"/>
</icu_chain>
```

Zebra is shipped with a field types file `icu.idx` which is an ICU chain version of `default.idx`.

Example 10.3 MARCXML indexing using ICU

The directory `examples/marcxml` includes a complete sample with MARCXML records that are DOM XML indexed using ICU chain rules. Study the `README` in the `marcxml` directory for details.

Chapter 11

Reference

The material in this chapter is drawn directly from the individual manual entries.

11.1 zebraidx

zebraidx — Zebra Administrative Tool

Synopsis

`zebraidx` [`-t` *type*] [`-c` *config*] [`-g` *group*] [`-d` *database*] [`-m` *mbytes*] [`-n`] [`-s`] [`-v` *level*] [`-l` *file*] [`-L`] [`-f` *number*] [`-v`] *command* [*file*...]

DESCRIPTION

zebraidx allows you to insert, delete or updates records in Zebra. **zebraidx** accepts a set options (see below) and exactly one command (mandatory).

COMMANDS

update *`directory`* Update the register with the files contained in *directory*. If no directory is provided, a list of files is read from `stdin`. See Administration in the Zebra Manual.

delete *`directory`* Remove the records corresponding to the files found under *directory* from the register.

adelete *`directory`* Remove the records corresponding to the files found under *directory* from the register. Unlike command `delete` this command does not fail if a record does not exist (but which is attempted deleted).

commit Write the changes resulting from the last `update` commands to the register. This command is only available if the use of shadow register files is enabled (see Shadow Registers in the Zebra Manual).

clean Clean shadow files and "forget" changes.

create ***database*** Create database.

drop ***database*** Drop database (delete database).

init Deletes an entire register (all files in shadow+register areas).

OPTIONS

-t ***type*** Update all files as *type*. Currently, the types supported are `text`, `alvis` and `grs.`*subtype*. Generally, it is probably advisable to specify the record types in the `zebra.cfg` file (see Record Types in the Zebra manual), to avoid confusion at subsequent updates.

-c ***config-file*** Read the configuration file *config-file* instead of `zebra.cfg`.

-g ***group*** Update the files according to the group settings for *group* (see Zebra Configuration File in the Zebra manual).

-d ***database*** The records located should be associated with the database name *database* for access through the Z39.50 server.

-l ***file*** Write log messages to *file* instead of `stderr`.

-m ***mbytes*** Use *mbytes* of memory before flushing keys to background storage. This setting affects performance when updating large databases.

-L Makes zebraidx skip symbolic links. By default, zebraidx follows them.

-n Disable the use of shadow registers for this operation (see Shadow Registers in the Zebra manual).

-s Show analysis of the indexing process. The maintenance program works in a read-only mode and doesn't change the state of the index. This options is very useful when you wish to test a new profile.

-V Show Zebra version.

-v ***level*** Set the log level to *level*. *level* should be one of `none`, `debug`, and `all`.

FILES

`zebra.cfg`

SEE ALSO

zebrasrv(8)

11.2 zebrasrv

zebrasrv — Zebra Server

Synopsis

`zebrasrv` [`-install`] [`-installa`] [`-remove`] [`-a` *file*] [`-v` *level*] [`-l` *file*] [`-u` *uid*] [`-c` *config*] [`-f` *vconfig*] [`-C` *fname*] [`-t` *minutes*] [`-k` *kilobytes*] [`-d` *daemon*] [`-w` *dir*] [`-p` *pidfile*] [`-ziDST1`] [listener-spec...]

DESCRIPTION

Zebra is a high-performance, general-purpose structured text indexing and retrieval engine. It reads structured records in a variety of input formats (e.g. email, XML, MARC) and allows access to them through exact boolean search expressions and relevance-ranked free-text queries.

zebrasrv is the Z39.50 and SRU frontend server for the **Zebra** search engine and indexer.

On Unix you can run the **zebrasrv** server from the command line - and put it in the background. It may also operate under the inet daemon. On WIN32 you can run the server as a console application or as a WIN32 Service.

OPTIONS

The options for **zebrasrv** are the same as those for YAZ' **yaz-ztest**. Option `-c` specifies a Zebra configuration file - if omitted `zebra.cfg` is read.

`-a` ***`file`*** Specify a file for dumping PDUs (for diagnostic purposes). The special name - (dash) sends output to `stderr`.

`-S` Don't fork or make threads on connection requests. This is good for debugging, but not recommended for real operation: Although the server is asynchronous and non-blocking, it can be nice to keep a software malfunction (okay then, a crash) from affecting all current users. The server can only accept a single connection in this mode.

`-1` Like `-S` but after one session the server exits. This mode is for debugging *only*.

`-T` Operate the server in threaded mode. The server creates a thread for each connection rather than a fork a process. Only available on UNIX systems that offers POSIX threads.

`-s` Use the SR protocol (obsolete).

`-z` Use the Z39.50 protocol (default). This option and `-s` complement each other. You can use both multiple times on the same command line, between listener-specifications (see below). This way, you can set up the server to listen for connections in both protocols concurrently, on different local ports.

`-l` ***`file`*** Specify an output file for the diagnostic messages. The default is to write this information to `stderr`

-c ***config-file*** Read configuration information from *config-file*. The default configuration is `./zebra.cfg`

-f ***vconfig*** This specifies an XML file that describes one or more YAZ frontend virtual servers. See section VIRTUAL HOSTS for details.

-C ***fname*** Sets SSL certificate file name for server (PEM).

-v ***level*** The log level. Use a comma-separated list of members of the set {fatal,debug,warn,log,malloc,all,none

-u ***uid*** Set user ID. Sets the real UID of the server process to that of the given user. It's useful if you aren't comfortable with having the server run as root, but you need to start it as such to bind a privileged port.

-w ***working-directory*** The server changes to this working directory during before listening on incoming connections. This option is useful when the server is operating from the inetd daemon (see `-i`).

-p ***pidfile*** Specifies that the server should write its Process ID to file given by *pidfile*. A typical location would be `/var/run/zebrasrv.pid`.

-i Use this to make the the server run from the inetd server (UNIX only). Make sure you use the logfile option `-l` in conjunction with this mode and specify the `-l` option before any other options.

-D Use this to make the server put itself in the background and run as a daemon. If neither `-i` nor `-D` is given, the server starts in the foreground.

-install Use this to install the server as an NT service (Windows NT/2000/XP only). Control the server by going to the Services in the Control Panel.

-installa Use this to install and activate the server as an NT service (Windows NT/2000/XP only). Control the server by going to the Services in the Control Panel.

-remove Use this to remove the server from the NT services (Windows NT/2000/XP only).

-t ***minutes*** Idle session timeout, in minutes. Default is 60 minutes.

-k ***size*** Maximum record size/message size, in kilobytes. Default is 1024 KB (1 MB).

-d ***daemon*** Set name of daemon to be used in hosts access file. See hosts_access(5) and tcpd(8).

A *listener-address* consists of an optional transport mode followed by a colon (:) followed by a listener address. The transport mode is either a file system socket `unix`, a SSL TCP/IP socket `ssl`, or a plain TCP/IP socket `tcp` (default).

For TCP, an address has the form

```
hostname | IP-number [: portnumber]
```

The port number defaults to 210 (standard Z39.50 port) for privileged users (root), and 9999 for normal users. The special hostname "@" is mapped to the address INADDR_ANY, which causes the server to listen on any local interface.

The default behavior for `zebrasrv` - if started as non-privileged user - is to establish a single TCP/IP listener, for the Z39.50 protocol, on port 9999.

```
zebrasrv @
zebrasrv tcp:some.server.name.org:1234
zebrasrv ssl:@:3000
```

To start the server listening on the registered port for Z39.50, or on a filesystem socket, and to drop root privileges once the ports are bound, execute the server like this from a root shell:

```
zebrasrv -u daemon @
zebrasrv -u daemon tcp:@:210
zebrasrv -u daemon unix:/some/file/system/socket
```

Here `daemon` is an existing user account, and the unix socket `/some/file/system/socket` is readable and writable for the `daemon` account.

Z39.50 Protocol Support and Behavior

Z39.50 Initialization

During initialization, the server will negotiate to version 3 of the Z39.50 protocol, and the option bits for Search, Present, Scan, NamedResultSets, and concurrentOperations will be set, if requested by the client. The maximum PDU size is negotiated down to a maximum of 1 MB by default.

Z39.50 Search

The supported query type are 1 and 101. All operators are currently supported with the restriction that only proximity units of type "word" are supported for the proximity operator. Queries can be arbitrarily complex. Named result sets are supported, and result sets can be used as operands without limitations. Searches may span multiple databases.

The server has full support for piggy-backed retrieval (see also the following section).

Z39.50 Present

The present facility is supported in a standard fashion. The requested record syntax is matched against the ones supported by the profile of each record retrieved. If no record syntax is given, SUTRS is the default. The requested element set name, again, is matched against any provided by the relevant record profiles.

Z39.50 Scan

The attribute combinations provided with the termListAndStartPoint are processed in the same way as operands in a query (see above). Currently, only the term and the globalOccurrences are returned with the termInfo structure.

Z39.50 Sort

Z39.50 specifies three different types of sort criteria. Of these Zebra supports the attribute specification type in which case the use attribute specifies the "Sort register". Sort registers are created for those fields that are of type "sort" in the default.idx file. The corresponding character mapping file in default.idx specifies the ordinal of each character used in the actual sort.

Z39.50 allows the client to specify sorting on one or more input result sets and one output result set. Zebra supports sorting on one result set only which may or may not be the same as the output result set.

Z39.50 Close

If a Close PDU is received, the server will respond with a Close PDU with reason=FINISHED, no matter which protocol version was negotiated during initialization. If the protocol version is 3 or more, the server will generate a Close PDU under certain circumstances, including a session timeout (60 minutes by default), and certain kinds of protocol errors. Once a Close PDU has been sent, the protocol association is considered broken, and the transport connection will be closed immediately upon receipt of further data, or following a short timeout.

Z39.50 Explain

Zebra maintains a "classic" Z39.50 Explain database on the side. This database is called `IR-Explain-1` and can be searched using the attribute set `exp-1`.

The records in the explain database are of type `grs.sgml`. The root element for the Explain grs.sgml records is `explain`, thus `explain.abs` is used for indexing.

Note

Zebra *must* be able to locate `explain.abs` in order to index the Explain records properly. Zebra will work without it but the information will not be searchable.

The SRU Server

In addition to Z39.50, Zebra supports the more recent and web-friendly IR protocol SRU. SRU can be carried over SOAP or a REST-like protocol that uses HTTP GET or POST to request search responses. The request itself is made of parameters such as `query`, `startRecord`, `maximumRecords` and `recordSchema`; the response is an XML document containing hit-count, result-set records, diagnostics, etc. SRU can be thought of as a re-casting of Z39.50 semantics in web-friendly terms; or as a standardisation of the ad-hoc query parameters used by search engines such as Google and AltaVista; or as a superset of A9's OpenSearch (which it predates).

Zebra supports Z39.50, SRU GET, SRU POST, SRU SOAP (SRW) - on the same port, recognising what protocol is used by each incoming requests and handling them accordingly. This is a achieved through the use of Deep Magic; civilians are warned not to stand too close.

Running zebrasrv as an SRU Server

Because Zebra supports all protocols on one port, it would seem to follow that the SRU server is run in the same way as the Z39.50 server, as described above. This is true, but only in an uninterestingly vacuous way: a Zebra server run in this manner will indeed recognise and accept SRU requests; but since it doesn't know how to handle the CQL queries that these protocols use, all it can do is send failure responses.

Note

It is possible to cheat, by having SRU search Zebra with a PQF query instead of CQL, using the `x-pquery` parameter instead of `query`. This is a **non-standard extension** of CQL, and a **very naughty** thing to do, but it does give you a way to see Zebra serving SRU ``right out of the box". If you start your favourite Zebra server in the usual way, on port 9999, then you can send your web browser to:

```
http://localhost:9999/Default?version=1.1
 &operation=searchRetrieve
 &x-pquery=mineral
 &startRecord=1
 &maximumRecords=1
```

This will display the XML-formatted SRU response that includes the first record in the result-set found by the query `mineral`. (For clarity, the SRU URL is shown here broken across lines, but the lines should be joined together to make single-line URL for the browser to submit.)

In order to turn on Zebra's support for CQL queries, it's necessary to have the YAZ generic front-end (which Zebra uses) translate them into the Z39.50 Type-1 query format that is used internally. And to do this, the generic front-end's own configuration file must be used. See the section called "YAZ server virtual hosts"; the salient point for SRU support is that **zebrasrv** must be started with the `-f frontendConfigFile` option rather than the `-c zebraConfigFile` option, and that the front-end configuration file must include both a reference to the Zebra configuration file and the CQL-to-PQF translator configuration file.

A minimal front-end configuration file that does this would read as follows:

```
<yazgfs>
  <server>
    <config>zebra.cfg</config>
    <cql2rpn>../../tab/pqf.properties</cql2rpn>
  </server>
</yazgfs>
```

The `<config>` element contains the name of the Zebra configuration file that was previously specified by the `-c` command-line argument, and the `<cql2rpn>` element contains the name of the CQL properties file specifying how various CQL indexes, relations, etc. are translated into Type-1 queries.

A zebra server running with such a configuration can then be queried using proper, conformant SRU URLs with CQL queries:

```
http://localhost:9999/Default?version=1.1
 &operation=searchRetrieve
 &query=title=utah and description=epicent*
```

```
    &startRecord=1
    &maximumRecords=1
```

SRU Protocol Support and Behavior

Zebra running as an SRU server supports SRU version 1.1, including CQL version 1.1. In particular, it provides support for the following elements of the protocol.

SRU Search and Retrieval

Zebra supports the SRU searchRetrieve operation.

One of the great strengths of SRU is that it mandates a standard query language, CQL, and that all conforming implementations can therefore be trusted to correctly interpret the same queries. It is with some shame, then, that we admit that Zebra also supports an additional query language, our own Prefix Query Format (PQF). A PQF query is submitted by using the extension parameter `x-pquery`, in which case the `query` parameter must be omitted, which makes the request not valid SRU. Please feel free to use this facility within your own applications; but be aware that it is not only non-standard SRU but not even syntactically valid, since it omits the mandatory `query` parameter.

SRU Scan

Zebra supports SRU scan operation. Scanning using CQL syntax is the default, where the standard `scanClause` parameter is used.

In addition, a mutant form of SRU scan is supported, using the non-standard `x-pScanClause` parameter in place of the standard `scanClause` to scan on a PQF query clause.

SRU Explain

Zebra supports SRU explain.

The ZeeRex record explaining a database may be requested either with a fully fledged SRU request (with `operation=explain` and version-number specified) or with a simple HTTP GET at the server's base-name. The ZeeRex record returned in response is the one embedded in the YAZ Frontend Server configuration file that is described in the the section called "YAZ server virtual hosts".

Unfortunately, the data found in the CQL-to-PQF text file must be added by hand-craft into the explain section of the YAZ Frontend Server configuration file to be able to provide a suitable explain record. Too bad, but this is all extreme new alpha stuff, and a lot of work has yet to be done ..

There is no linkage whatsoever between the Z39.50 explain model and the SRU explain response (well, at least not implemented in Zebra, that is ..). Zebra does not provide a means using Z39.50 to obtain the ZeeRex record.

Other SRU operations

In the Z39.50 protocol, Initialization, Present, Sort and Close are separate operations. In SRU, however, these operations do not exist.

- SRU has no explicit initialization handshake phase, but commences immediately with searching, scanning and explain operations.
- Neither does SRU have a close operation, since the protocol is stateless and each request is self-contained. (It is true that multiple SRU request/response pairs may be implemented as multiple HTTP request/response pairs over a single persistent TCP/IP connection; but the closure of that connection is not a protocol-level operation.)
- Retrieval in SRU is part of the `searchRetrieve` operation, in which a search is submitted and the response includes a subset of the records in the result set. There is no direct analogue of Z39.50's Present operation which requests records from an established result set. In SRU, this is achieved by sending a subsequent `searchRetrieve` request with the query `cql.resultSetId=`*id* where *id* is the identifier of the previously generated result-set.
- Sorting in CQL is done within the `searchRetrieve` operation - in v1.1, by an explicit `sort` parameter, but the forthcoming v1.2 or v2.0 will most likely use an extension of the query language, CQL sorting.

It can be seen, then, that while Zebra operating as an SRU server does not provide the same set of operations as when operating as a Z39.50 server, it does provide equivalent functionality.

SRU Examples

Surf into `http://localhost:9999` to get an explain response, or use

```
http://localhost:9999/?version=1.1&operation=explain
```

See number of hits for a query

```
http://localhost:9999/?version=1.1&operation=searchRetrieve
&query=text=(plant%20and%20soil)
```

Fetch record 5-7 in Dublin Core format

```
http://localhost:9999/?version=1.1&operation=searchRetrieve
                  &query=text=(plant%20and%20soil)
                  &startRecord=5&maximumRecords=2&recordSchema=dc
```

Even search using PQF queries using the *extended naughty parameter* `x-pquery`

```
 http://localhost:9999/?version=1.1&operation=searchRetrieve
                  &x-pquery=@attr%201=text%20@and%20plant%20soil
```

Or scan indexes using the *extended extremely naughty parameter* `x-pScanClause`

```
 http://localhost:9999/?version=1.1&operation=scan
                  &x-pScanClause=@attr%201=text%20something
```

Don't do this in production code! But it's a great fast debugging aid.

YAZ server virtual hosts

The Virtual hosts mechanism allows a YAZ frontend server to support multiple backends. A backend is selected on the basis of the TCP/IP binding (port+listening address) and/or the virtual host.

A backend can be configured to execute in a particular working directory. Or the YAZ frontend may perform CQL to RPN conversion, thus allowing traditional Z39.50 backends to be offered as a SRU service. SRU Explain information for a particular backend may also be specified.

For the HTTP protocol, the virtual host is specified in the Host header. For the Z39.50 protocol, the virtual host is specified as in the Initialize Request in the OtherInfo, OID 1.2.840.10003.10.1000.81.1.

Note
Not all Z39.50 clients allows the VHOST information to be set. For those the selection of the backend must rely on the TCP/IP information alone (port and address).

The YAZ frontend server uses XML to describe the backend configurations. Command-line option `-f` specifies filename of the XML configuration.

The configuration uses the root element `yazgfs`. This element includes a list of `listen` elements, followed by one or more `server` elements.

The `listen` describes listener (transport end point), such as TCP/IP, Unix file socket or SSL server. Content for a listener:

CDATA (required) The CDATA for the `listen` element holds the listener string, such as `tcp:@:210`, `tcp:server1:2100`, etc.

attribute `id` (optional) identifier for this listener. This may be referred to from server sections.

Note
We expect more information to be added for the listen section in a future version, such as CERT file for SSL servers.

The `server` describes a server and the parameters for this server type. Content for a server:

attribute `id` (optional) Identifier for this server. Currently not used for anything, but it might be for logging purposes.

attribute `listenref` (optional) Specifies listener for this server. If this attribute is not given, the server is accessible from all listener. In order for the server to be used for real, however, the virtual host must match (if specified in the configuration).

element `config` (optional) Specifies the server configuration. This is equivalent to the config specified using command line option `-c`.

element `directory` (optional) Specifies a working directory for this backend server. If specified, the YAZ frontend changes current working directory to this directory whenever a backend of this type is started (backend handler bend_start), stopped (backend handler hand_stop) and initialized (bend_init).

element `host` (optional) Specifies the virtual host for this server. If this is specified a client *must* specify this host string in order to use this backend.

element `cql2rpn` (optional) Specifies a filename that includes CQL to RPN conversion for this backend server. See CQL section in YAZ manual. If given, the backend server will only "see" a Type-1/RPN query.

element `explain` (optional) Specifies SRU ZeeRex content for this server - copied verbatim to the client. As things are now, some of the Explain content seems redundant because host information, etc. is also stored elsewhere.

The format of the Explain record is described in detail, with examples, on the file at the ZeeRex web-site.

The XML below configures a server that accepts connections from two ports, TCP/IP port 9900 and a local UNIX file socket. We name the TCP/IP server `public` and the other server `internal`.

```
<yazgfs>
 <listen id="public">tcp:@:9900</listen>
 <listen id="internal">unix:/var/tmp/socket</listen>
 <server id="server1">
   <host>server1.mydomain</host>
   <directory>/var/www/s1</directory>
   <config>config.cfg</config>
 </server>
 <server id="server2">
   <host>server2.mydomain</host>
   <directory>/var/www/s2</directory>
   <config>config.cfg</config>
   <cql2rpn>../etc/pqf.properties</cql2rpn>
   <explain xmlns="http://explain.z3950.org/dtd/2.0/">
     <serverInfo>
       <host>server2.mydomain</host>
       <port>9900</port>
       <database>a</database>
     </serverInfo>
   </explain>
 </server>
 <server id="server3" listenref="internal">
   <directory>/var/www/s3</directory>
   <config>config.cfg</config>
 </server>
</yazgfs>
```

There are three configured backend servers. The first two servers, `"server1"` and `"server2"`, can be reached by both listener addresses - since no `listenref` attribute is specified. In order to distinguish between the two a virtual host has been specified for each of server in the `host` elements.

For `"server2"` elements for CQL to RPN conversion is supported and explain information has been added (a short one here to keep the example small).

The third server, "server3" can only be reached via listener "internal".

SEE ALSO

zebraidx(1)

11.3 idzebra-config

idzebra-config — Script to get information about idzebra

Synopsis

idzebra-config [--prefix[=*DIR*]] [--version] [--libs] [--lalibs] [--cflags] [--tab] [--modules] [libraries...]

DESCRIPTION

idzebra-config is a script that returns information that your own software should use to build software that uses idzebra.

The following libraries are supported:

None

OPTIONS

--prefix[=*DIR*] Returns prefix of idzebra or assume a different one if DIR is specified.

--version Returns version of idzebra.

--libs Library specification be used when linking with idzebra.

--lalibs Return library specification.

--cflags Return C Compiler flags.

--tab Return directory of idzebra tables.

--modules Return directory for Zebra modules.

FILES

/usr/local/bin/idzebra-config-2.0

/usr/local/lib/libidzebra*2.0.a

/usr/local/include/idzebra-2.0/idzebra/*.h

Appendix A

License

Zebra Server, Copyright © 1994-2009 Index Data ApS.

Zebra is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

Zebra is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Zebra; see the file LICENSE.zebra. If not, write to the Free Software Foundation, 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Appendix B

GNU General Public License

B.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and

2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

B.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

B.2.1 Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The 'Program', below, refers to any such program or work, and a 'work based on the Program' means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term 'modification'.) Each licensee is addressed as 'you'.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

B.2.2 Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

B.2.3 Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that

you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

B.2.4 Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

B.2.5 Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

B.2.6 Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

B.2.7 Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

B.2.8 Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

B.2.9 Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

B.2.10 Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and 'any later version', you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

B.2.11 Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

B.2.12 NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM 'AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

B.2.13 Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING

ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

B.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the 'copyright' line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a 'copyright disclaimer' for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix C

About Index Data and the Zebra Server

Index Data is a consulting and software-development enterprise that specializes in library and information management systems. Our interests and expertise span a broad range of related fields, and one of our primary, long-term objectives is the development of a powerful information management system with open network interfaces and hyper-media capabilities.

We make this software available free of charge, on a fairly unrestrictive license; as a service to the networking community, and to further the development of quality software for open network communication.

We'll be happy to answer questions about the software, and about ourselves in general.

```
Index Data ApS
Købmagergade 43, 2.
1150 Copenhagen K
Denmark
Phone +45 3341 0100
Fax +45 3341 0101
Email info@indexdata.dk
```

The *Random House College Dictionary*, 1975 edition offers this definition of the word "Zebra":

Zebra, n., any of several horselike, African mammals of the genus Equus, having a characteristic pattern of black or dark-brown stripes on a whitish background.